Tell us about your PDF experience.

C++ Standard Library reference (STL)

Article • 08/17/2022

A C++ program can call on a large number of functions from this conforming implementation of the C++ Standard Library. These functions perform services such as input and output and provide efficient implementations of frequently used operations.

For more information about linking with the appropriate Visual C++ runtime `.lib` file, see C runtime (CRT) and C++ Standard Library (STL) .lib files.

> **Note**
>
> Microsoft's implementation of the C++ Standard Library is often referred to as the *STL* or *Standard Template Library*. Although *C++ Standard Library* is the official name of the library as defined in ISO 14882, due to the popular use of "STL" and "Standard Template Library" in search engines, we occasionally use those names to make it easier to find our documentation.

From a historical perspective, "STL" originally referred to the Standard Template Library written by Alexander Stepanov. Parts of that library were standardized in the C++ Standard Library, along with the ISO C runtime library, parts of the Boost library, and other functionality. Sometimes "STL" is used to refer to the containers and algorithms parts of the C++ Standard Library adapted from Stepanov's STL. In this documentation, Standard Template Library (STL) refers to the C++ Standard Library as a whole.

In this section

C++ Standard Library overview Provides an overview of the Microsoft implementation of the C++ Standard Library.

iostream programming Provides an overview of `iostream` programming.

Header files reference Provides links to reference topics about the C++ Standard Library header files, with code examples.

C++ standard library header files

Article • 08/03/2021

Header files for the C++ standard library and extensions, by category.

Headers by category

Category	Headers
Algorithms	<algorithm>, <cstdlib>, <numeric>
Atomic operations	<atomic>[11]
C library wrappers	<cassert>, <ccomplex>[11 a b], <cctype>, <cerrno>, <cfenv>[11], <cfloat>, <cinttypes>[11], <ciso646>[b], <climits>, <clocale>, <cmath>, <csetjmp>, <csignal>, <cstdalign>[11 a b], <cstdarg>, <cstdbool>[11 a b], <cstddef>, <cstdint>[11], <cstdio>, <cstdlib>, <cstring>, <ctgmath>[11 a b], <ctime>, <cuchar>[11], <cwchar>, <cwctype>
Concepts	`<concepts>`[20]
Containers	
Sequence containers	<array>[11], <deque>, <forward_list>[11], <list>, <vector>
Ordered associative containers	<map>, <set>
Unordered associative containers	<unordered_map>[11], <unordered_set>[11]
Container adaptors	<queue>, <stack>
Container views	[20]
Errors and exception handling	<cassert>, <exception>, <stdexcept>, <system_error>[11]
General utilities	`<any>`[17], <bit>[20], <bitset>, <cstdlib>, `<execution>`[17], <functional>, <memory>, `<memory_resource>`[17], `<optional>`[17], <ratio>[11], <scoped_allocator>[11], <tuple>[11], <type_traits>[11], <typeindex>[11], <utility>, `<variant>`[17]

Category	Headers
I/O and formatting	<cinttypes>[11], <cstdio>, <filesystem>[17], <fstream>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <ostream>, <sstream>, <streambuf>, <strstream>[c], `<syncstream>`[20]
Iterators	<iterator>
Language support	<cfloat>, <climits>, <codecvt>[11 a], `<compare>`[20], `<contract>`[20], `<coroutine>`[20], <csetjmp>, <csignal>, <cstdarg>, <cstddef>, <cstdint>[11], <cstdlib>, <exception>, <initializer_list>[11], <limits>, <new>, <typeinfo>, `<version>`[20]
Localization	<clocale>, <codecvt>[11 a], <cvt/wbuffer>, <cvt/wstring>, <locale>
Math and numerics	`<bit>`[20], <cfenv>[11], <cmath>, <complex>, <cstdlib>, <limits>, <numeric>, <random>[11], <ratio>[11], <valarray>
Memory management	<allocators>, <memory>, `<memory_resource>`[17], <new>, <scoped_allocator>[11]
Multithreading	<atomic>[11], <condition_variable>[11], <future>[11], <mutex>[11], <shared_mutex>[14], <thread>[11]
Ranges	`<ranges>`[20]
Regular expressions	<regex>[11]
Strings and character data	<charconv>[17], <cctype>, <cstdlib>, <cstring>, <cuchar>[11], <cwchar>, <cwctype>, <regex>[11], <string>, <string_view>[17]
Time	<chrono>[11], <ctime>

[11] Added in the C++11 standard.
[14] Added in the C++14 standard.
[17] Added in the C++17 standard.
[20] Added in the draft C++20 standard.
[a] Deprecated in the C++17 standard.
[b] Removed in the draft C++20 standard.
[c] Deprecated in the C++98 standard.

See also

Using C++ library headers
C++ standard library

`<algorithm>`

Article • 01/30/2023

Defines C++ Standard Library container template functions that perform algorithms.

Syntax

```
C++

(see links below for specific algorithm syntax)
```

> **Note**
>
> The `<algorithm>` library also uses the `#include <initializer_list>` statement.

Remarks

The C++ Standard Library algorithms can operate on various data structures. The data structures that they can operate on include not only the C++ Standard Library container classes such as `vector` and `list`, but also user-defined data structures and arrays of elements, as long as they satisfy the requirements of a particular algorithm. C++ Standard Library algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators.

C++ Standard Library algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all iterators in the ranges must be dereferenceable and, within the sequences of each range, the last position must be reachable from the first by incrementing the iterator.

Starting in C++20, most of the algorithms defined in <algorithm> are also available in a form that takes a `range`. For example, rather than call `sort(v1.begin(), v1.end(), greater<int>());`, you can call `ranges::sort(v1, greater<int>());`

The C++ Standard Library algorithms can work with different types of container objects at the same time. Two suffixes have been used to convey information about the purpose of the algorithms:

- The `_if` suffix indicates that the algorithm is used with function objects that operate on the values of the elements rather than on the elements themselves. For example, the `find_if` algorithm looks for elements whose values satisfy the criterion specified by a function object, whereas the `find` algorithm searches for a particular value.

- The `_copy` suffix indicates that the algorithm generally modifies copied values rather than copy modified values. In other words, they don't modify the source range's elements but put the results into an output range/iterator. For example, the `reverse` algorithm reverses the order of the elements within a range, whereas the `reverse_copy` algorithm copies the reversed result into a destination range.

C++ Standard Library algorithms are often classified into groups to indicate their purpose or requirements. These include modifying algorithms that change the value of elements as compared with non-modifying algorithms that don't. Mutating algorithms change the order of elements, but not the values of their elements. Removing algorithms can eliminate elements from a range or a copy of a range. Sorting algorithms reorder the elements in a range in various ways and sorted range algorithms only act on ranges whose elements have been sorted in a particular way.

The C++ Standard Library numeric algorithms that are provided for numerical processing have their own header file <numeric>, and function objects and adaptors are defined in the header <functional>. Function objects that return Boolean values are known as predicates. The default binary predicate is the comparison `operator<`. In general, the elements being ordered need to be less than comparable so that, given any two elements, it can be determined either that they're equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering among the nonequivalent elements.

Algorithms

Name	Description
adjacent_find	Searches for two adjacent elements that are either equal or satisfy a specified condition.
all_of	Returns `true` when a condition is present at each element in the given range.
any_of	Returns `true` when a condition is present at least once in the specified range of elements.

Name	Description
binary_search	Tests whether there is an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.
clamp	
copy	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.
copy_backward	Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.
copy_if	Copy all elements in a given range that test `true` for a specified condition
copy_n	Copies a specified number of elements.
count	Returns the number of elements in a range whose values match a specified value.
count_if	Returns the number of elements in a range whose values match a specified condition.
equal	Compares two ranges element by element either for equality or equivalence in a sense specified by a binary predicate.
equal_range	Finds a pair of positions in an ordered range, the first less than or equivalent to the position of a specified element and the second greater than the element's position, where the sense of equivalence or ordering used to establish the positions in the sequence may be specified by a binary predicate.
fill	Assigns the same new value to every element in a specified range.
fill_n	Assigns a new value to a specified number of elements in a range starting with a particular element.
find	Locates the position of the first occurrence of an element in a range that has a specified value.
find_end	Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.

Name	Description
find_first_of	Searches for the first occurrence of any of several values within a target range or for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.
find_if	Locates the position of the first occurrence of an element in a range that satisfies a specified condition.
find_if_not	Returns the first element in the indicated range that doesn't satisfy a condition.
for_each	Applies a specified function object to each element in a forward order within a range and returns the function object.
for_each_n	
generate	Assigns the values generated by a function object to each element in a range.
generate_n	Assigns the values generated by a function object to a specified number of elements in a range and returns to the position one past the last assigned value.
includes	Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.
inplace_merge	Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.
is_heap	Returns `true` if the elements in the specified range form a heap.
is_heap_until	Returns `true` if the specified range forms a heap until the last element.
is_partitioned	Returns `true` if all the elements in the given range that test `true` for a condition come before any elements that test `false`.
is_permutation	Determines whether the elements in a given range form a valid permutation.
is_sorted	Returns `true` if the elements in the specified range are in sorted order.
is_sorted_until	Returns `true` if the elements in the specified range are in sorted order.
iter_swap	Exchanges two values referred to by a pair of specified iterators.

Name	Description
lexicographical_compare	Compares element by element between two sequences to determine which is lesser of the two.
lower_bound	Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value, where the ordering criterion may be specified by a binary predicate.
make_heap	Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.
max	Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.
max_element	Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.
merge	Combines all the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
min	Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.
min_element	Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.
minmax	Compares two input parameters and returns them as a pair, in order of least to greatest.
minmax_element	Performs the work performed by min_element and max_element in one call.
mismatch	Compares two ranges element by element either for equality or equivalent in a sense specified by a binary predicate and locates the first position where a difference occurs.
<alg> move	Move elements associated with a specified range.
move_backward	Moves the elements of one iterator to another. The move starts with the last element in a specified range, and ends with the first element in that range.
next_permutation	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
none_of	Returns `true` when a condition is never present among elements in the given range.

Name	Description
nth_element	Partitions a range of elements, correctly locating the *n*th element of the sequence in the range so that all the elements in front of it are less than or equal to it and all the elements that follow it in the sequence are greater than or equal to it.
partial_sort	Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.
partial_sort_copy	Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.
partition	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it.
partition_copy	Copies elements for which a condition is `true` to one destination, and for which the condition is `false` to another. The elements must come from a specified range.
partition_point	Returns the first element in the given range that doesn't satisfy the condition. The elements are sorted so that those that satisfy the condition come before those that don't.
pop_heap	Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.
prev_permutation	Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists, where the sense of next may be specified with a binary predicate.
push_heap	Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.
random_shuffle	Rearranges a sequence of *N* elements in a range into one of *N*! possible arrangements selected at random.
remove	Eliminates a specified value from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
remove_copy	Copies elements from a source range to a destination range, except that elements of a specified value aren't copied, without disturbing the order of the remaining elements and returning the end of a new destination range.

Name	Description
remove_copy_if	Copies elements from a source range to a destination range, except that satisfying a predicate aren't copied, without disturbing the order of the remaining elements and returning the end of a new destination range.
remove_if	Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements and returning the end of a new range free of the specified value.
replace	Examines each element in a range and replaces it if it matches a specified value.
replace_copy	Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.
replace_copy_if	Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.
replace_if	Examines each element in a range and replaces it if it satisfies a specified predicate.
reverse	Reverses the order of the elements within a range.
reverse_copy	Reverses the order of the elements within a source range while copying them into a destination range
rotate	Exchanges the elements in two adjacent ranges.
rotate_copy	Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.
sample	
search	Searches for the first occurrence of a sequence within a target range whose elements are equal to those in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.
search_n	Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.
set_difference	Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

Name	Description
set_intersection	Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_symmetric_difference	Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
set_union	Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.
sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.
shuffle	Shuffles (rearranges) elements for a given range using a random number generator.
sort_heap	Converts a heap into a sorted range.
stable_partition	Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding those that fail to satisfy it, preserving the relative order of equivalent elements.
stable_sort	Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate and preserves the relative ordering of equivalent elements.
swap	Exchanges the values of the elements between two types of objects, assigning the contents of the first object to the second object and the contents of the second to the first.
swap_ranges	Exchanges the elements of one range with the elements of another, equal sized range.
transform	Applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.
unique	Removes duplicate elements that are next to each other in a specified range.
unique_copy	Copies elements from a source range into a destination range except for the duplicate elements that are next to each other.
upper_bound	Finds the position of the first element in an ordered range that has a value that is greater than a specified value, where the ordering criterion may be specified by a binary predicate.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<algorithm>` functions

Article • 11/07/2022

`adjacent_find`

Searches for two adjacent elements that are either equal or satisfy a specified condition.

C++

```
template<class ForwardIterator>
ForwardIterator adjacent_find(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator , class BinaryPredicate>
ForwardIterator adjacent_find(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator adjacent_find(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class
BinaryPredicate>
ForwardIterator adjacent_find(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator at the position of the first element in the range to be searched.

last

A forward iterator at the position one past the final element in the range to be searched.

pred

The binary predicate giving the condition to be satisfied by the values of the adjacent elements in the range being searched.

Return value

A forward iterator to the first of the adjacent elements that are either equal to each other (in the first version) or that satisfy the condition given by the binary predicate (in the second version) if such a pair of elements is found. Otherwise, an iterator pointing to *last* is returned.

Remarks

The `adjacent_find` algorithm is a nonmutating sequence algorithm. The range to be searched must be valid. All pointers must be dereferenceable, and the last position must be reachable from the first by incrementation. The time complexity of the algorithm is linear in the number of elements contained in the range.

The `operator==` used to determine the match between elements must impose an equivalence relation between its operands.

Example

C++

```
// alg_adj_fnd.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

// Returns whether second element is twice the first
bool twice (int elem1, int elem2 )
{
    return elem1 * 2 == elem2;
}

int main()
{
    using namespace std;
    list<int> L;
    list<int>::iterator Iter;
    list<int>::iterator result1, result2;

    L.push_back( 50 );
    L.push_back( 40 );
```

```
    L.push_back( 10 );
    L.push_back( 20 );
    L.push_back( 20 );

    cout << "L = ( " ;
    for ( Iter = L.begin( ) ; Iter != L.end( ) ; Iter++ )
        cout << *Iter << " ";
    cout << ")" << endl;

    result1 = adjacent_find( L.begin( ), L.end( ) );
    if ( result1 == L.end( ) )
        cout << "There are not two adjacent elements that are equal."
            << endl;
    else
        cout << "There are two adjacent elements that are equal.\n"
            << "They have a value of "
            << *( result1 ) << "." << endl;

    result2 = adjacent_find( L.begin( ), L.end( ), twice );
    if ( result2 == L.end( ) )
        cout << "There are not two adjacent elements where the "
            << "second is twice the first." << endl;
    else
    {
        cout << "There are two adjacent elements where "
            << "the second is twice the first.\n"
            << "They have values of " << *(result2++)
            << " & " << *result2 << "." << endl;
    }
}
```

Output

```
L = ( 50 40 10 20 20 )
There are two adjacent elements that are equal.
They have a value of 20.
There are two adjacent elements where the second is twice the first.
They have values of 10 & 20.
```

`all_of`

Returns `true` when a condition is present at each element in the given range.

C++

```
template<class InputIterator, class UnaryPredicate>
bool all_of(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);
```

```
template <class ExecutionPolicy, class ForwardIterator, class
UnaryPredicate>
bool all_of(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates where to start to check for a condition. The iterator marks where a range of elements starts.

last

An input iterator that indicates the end of the range of elements to check for a condition.

pred

A condition to test for. *pred* is a user-defined unary predicate function object that defines the condition to be satisfied by an element being checked. A unary predicate takes a single argument and returns **true** or **false**.

Return value

Returns **true** if the condition is detected at each element in the indicated range or if the range is empty, and **false** otherwise.

Remarks

The template function returns **true** only if, for each `N` in the range `[0, last - first)`, the predicate `pred(*(first + N))` is **true**.

Example

C++

```cpp
// alg_all_of.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;

    list<int> li { 50, 40, 10, 20, 20 };
    list<int>::iterator iter;

    cout << "li = ( ";
    for (iter = li.begin(); iter != li.end(); iter++)
        cout << *iter << " ";
    cout << ")" << endl;

    // Check if all elements in li are even.
    auto is_even = [](int elem){ return !(elem % 2); };
    if (all_of(li.begin(), li.end(), is_even))
        cout << "All the elements are even numbers.\n";
    else
        cout << "Not all the elements are even numbers.\n";
}
```

Output

```
li = ( 50 40 10 20 20 )
All the elements are even numbers.
```

any_of

Returns `true` when a condition is present at least once in the specified range of elements.

C++

```cpp
template<class InputIterator, class UnaryPredicate>
bool any_of(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template <class ExecutionPolicy, class ForwardIterator, class
UnaryPredicate>
bool any_of(
    ExecutionPolicy&& exec,
    ForwardIterator first,
```

```
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates where to start checking a range of elements for a condition.

last

An input iterator that indicates the end of the range of elements to check for a condition.

pred

A condition to test for. This test is provided by a user-defined predicate function object. The predicate defines the condition to be satisfied by the element being tested. A unary predicate takes a single argument and returns **true** or **false**.

Return value

Returns **true** if the condition is detected at least once in the indicated range, **false** if the condition is never detected.

Remarks

The template function returns **true** only if, for some `N` in the range

`[0, last - first)`, the predicate `pred(*(first + N))` is true.

Example

C++

```
// alg_any_of.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

int main()
```

```
{
    using namespace std;

    list<int> li { 51, 41, 11, 21, 20 };

    cout << "li = ( ";
    for (auto const& el : li)
        cout << el << " ";
    cout << ")" << endl;

    // Check if there's an even element in li.
    auto is_even = [](int const elem){ return !(elem % 2); };
    if (any_of(li.begin(), li.end(), is_even))
        cout << "There's an even element in li.\n";
    else
        cout << "There are no even elements in li.\n";
}
```

Output

```
li = ( 51 41 11 21 20 )
There's an even element in li.
```

`binary_search`

Tests whether there's an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.

C++

```
template<class ForwardIterator, class Type>
bool binary_search(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ForwardIterator, class Type, class BinaryPredicate>
bool binary_search(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    BinaryPredicate pred);
```

Parameters

`first`

A forward iterator addressing the position of the first element in the range to be

searched.

`last`
A forward iterator addressing the position one past the final element in the range to be searched.

`value`
The value required to be matched by the value of the element or that must satisfy the condition with the element value specified by the binary predicate.

`pred`
User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns `true` when satisfied and `false` when not satisfied.

Return value

`true` if an element is found in the range that is equal or equivalent to the specified value; otherwise, `false`.

Remarks

The sorted source range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The sorted range must each be arranged as a precondition to the application of the `binary_search` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The source ranges aren't modified by `binary_search`.

The value types of the forward iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements.

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (`last-first`).

Example

C++

```
// alg_bin_srch.cpp
// compile with: /EHsc
#include <list>
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    list<int> List1;

    List1.push_back( 50 );
    List1.push_back( 10 );
    List1.push_back( 30 );
    List1.push_back( 20 );
    List1.push_back( 25 );
    List1.push_back( 5 );

    List1.sort();

    cout << "List1 = ( " ;
    for ( auto Iter : List1 )
        cout << Iter << " ";
    cout << ")" << endl;

    // default binary search for 10
    if ( binary_search(List1.begin(), List1.end(), 10) )
        cout << "There is an element in list List1 with a value equal to
10."
        << endl;
    else
        cout << "There is no element in list List1 with a value equal to
10."
        << endl;

    // a binary_search under the binary predicate greater
    List1.sort(greater<int>());
    if ( binary_search(List1.begin(), List1.end(), 10, greater<int>()) )
        cout << "There is an element in list List1 with a value greater than
```

```
10 "
        << "under greater than." << endl;
    else
        cout << "No element in list List1 with a value greater than 10 "
        << "under greater than." << endl;

    // a binary_search under the user-defined binary predicate mod_lesser
    vector<int> v1;

    for ( auto i = -2; i <= 4; ++i )
    {
        v1.push_back(i);
    }

    sort(v1.begin(), v1.end(), mod_lesser);

    cout << "Ordered using mod_lesser, vector v1 = ( " ;
    for ( auto Iter : v1 )
        cout << Iter << " ";
    cout << ")" << endl;

    if ( binary_search(v1.begin(), v1.end(), -3, mod_lesser) )
        cout << "There is an element with a value equivalent to -3 "
        << "under mod_lesser." << endl;
    else
        cout << "There is not an element with a value equivalent to -3 "
        << "under mod_lesser." << endl;
}
```

Output

```
List1 = ( 5 10 20 25 30 50 )
There is an element in list List1 with a value equal to 10.
There is an element in list List1 with a value greater than 10 under greater
than.
Ordered using mod_lesser, vector v1 = ( 0 -1 1 -2 2 3 4 )
There is an element with a value equivalent to -3 under mod_lesser.
```

clamp

Compares a value to an upper and lower bound, and returns a reference to the value if it's between the bounds, or a reference to the upper or lower bound if the value is above or below them, respectively.

C++

```
template<class Type>
constexpr const Type& clamp(
    const Type& value,
```

```
    const Type& lower,
    const Type& upper);

template<class Type, class Compare>
constexpr const Type& clamp(
    const Type& value,
    const Type& lower,
    const Type& upper,
    Compare pred);
```

Parameters

value

The value to compare to *upper* and *lower*.

lower

The lower bound of values to clamp *value* to.

upper

The upper bound of values to clamp *value* to.

pred

A predicate used to compare *value* to *lower* or *upper*. A comparison predicate takes two arguments and returns **true** if the first is in some sense less than the second, and otherwise, **false**.

Return value

Returns a reference to *lower* if `value < lower`, or a reference to *upper* if `upper < value`. Otherwise, it returns a reference to *value*.

Remarks

The behavior is undefined if *upper* is less than *lower*.

copy

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.

```
C++
```

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator first,
    InputIterator last,
    OutputIterator destBeg);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 copy(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result);
```

Parameters

exec

The execution policy to use.

first

An input iterator addressing the position of the first element in the source range.

last

An input iterator addressing the position that is one past the final element in the source range.

destBeg

An output iterator addressing the position of the first element in the destination range.

Return value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses `result` + (*last* - *first*).

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

Because the algorithm copies the source elements in order beginning with the first element, the destination range can overlap with the source range provided the *last* position of the source range isn't contained in the destination range. `copy` can be used to shift elements to the left but not the right, unless there's no overlap between the

source and destination ranges. To shift to the right any number of positions, use the copy_backward algorithm.

The `copy` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It can't be used to create new elements and can't insert elements into an empty container directly.

Example

C++

```
// alg_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1.push_back( 10 * i );

    int ii;
    for ( ii = 0 ; ii <= 10 ; ii++ )
        v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy the first 3 elements of v1 into the middle of v2
    copy( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 4 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the left
    copy( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 2 );
```

```
    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;
}
```

Output

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```

copy_backward

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.

C++

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
    BidirectionalIterator1 first,
    BidirectionalIterator1 last,
    BidirectionalIterator2 destEnd);
```

Parameters

first

A bidirectional iterator addressing the position of the first element in the source range.

last

A bidirectional iterator addressing the position that is one past the final element in the source range.

destEnd

A bidirectional iterator addressing the position of one past the final element in the destination range.

Return value

An output iterator addressing the position that is one past the final element in the destination range, that is, the iterator addresses `destEnd - (last - first)`.

Remarks

The source range must be valid and there must be sufficient space at the destination to hold all the elements being copied.

The `copy_backward` algorithm imposes more stringent requirements than that the `copy` algorithm. Both its input and output iterators must be bidirectional.

The `copy_backward` and move_backward algorithms are the only C++ Standard Library algorithms designating the output range with an iterator pointing to the end of the destination range.

Because the algorithm copies the source elements in order beginning with the last element, the destination range can overlap with the source range provided the *`first`* position of the source range isn't contained in the destination range. `copy_backward` can be used to shift elements to the right but not the left, unless there's no overlap between the source and destination ranges. To shift to the left any number of positions, use the copy algorithm.

The `copy_backward` algorithm only modifies values pointed to by the iterators, assigning new values to elements in the destination range. It can't be used to create new elements and can't insert elements into an empty container directly.

Example

C++

```
// alg_copy_bkwd.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 5 ; ++i )
        v1.push_back( 10 * i );

    int ii;
```

```
    for ( ii = 0 ; ii <= 10 ; ++ii )
        v2.push_back( 3 * ii );

    cout << "v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; ++Iter1 )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To copy_backward the first 3 elements of v1 into the middle of v2
    copy_backward( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 7 );

    cout << "v2 with v1 insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // To shift the elements inserted into v2 two positions
    // to the right
    copy_backward( v2.begin( )+4, v2.begin( ) + 7, v2.begin( ) + 9 );

    cout << "v2 with shifted insert = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")" << endl;
}
```

Output

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 6 9 0 10 0 10 20 27 30 )
```

copy_if

In a range of elements, copies the elements that are `true` for the specified condition.

C++

```
template<class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator copy_if(
    InputIterator first,
    InputIterator last,
    OutputIterator dest,
    UnaryPredicate pred);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class UnaryPredicate>
ForwardIterator2 copy_if(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates the start of a range to check for the condition.

last

An input iterator that indicates the end of the range.

dest

The output iterator that indicates the destination for the copied elements.

pred

The condition against which every element in the range is tested. This condition is provided by a user-defined predicate function object. A unary predicate takes one argument and returns **`true`** or **`false`**.

Return value

An output iterator that equals *dest* incremented once for each element that fulfills the condition. In other words, the return value minus *dest* equals the number of copied elements.

Remarks

The template function evaluates

`if (pred(*first + N)) * dest++ = *(first + N))`

once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. If *dest* and *first* designate regions of storage, *dest*

must not be in the range `[ first, last )`.

Example

C++

```
// alg_copy_if.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

void listlist(std::list<int> l)
{
    std::cout << "( ";
    for (auto const& el : l)
        std::cout << el << " ";
    std::cout << ")" << std::endl;
}

int main()
{
    using namespace std;
    list<int> li{ 46, 59, 88, 72, 79, 71, 60, 5, 40, 84 };
    list<int> le(li.size()); // le = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    list<int> lo(li.size()); // lo = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    cout << "li = ";
    listlist(li);

    // is_even checks if the element is even.
    auto is_even = [](int const elem) { return !(elem % 2); };
    // use copy_if to select only even elements from li
    // and copy them to le, starting from le's begin position
    auto ec = copy_if(li.begin(),li.end(), le.begin(), is_even);
    le.resize(std::distance(le.begin(), ec));  // shrink le to new size

    cout << "Even numbers are le = ";
    listlist(le);

    // is_odd checks if the element is odd.
    auto is_odd = [](int const elem) { return (elem % 2); };
    // use copy_if to select only odd elements from li
    // and copy them to lo, starting from lo's begin position
    auto oc = copy_if(li.begin(), li.end(), lo.begin(), is_odd);
    lo.resize(std::distance(lo.begin(), oc));  // shrink lo to new size

    cout << "Odd numbers are lo = ";
    listlist(lo);
}
```

Output

```
li = ( 46 59 88 72 79 71 60 5 40 84 )
Even numbers are le = ( 46 88 72 60 40 84 )
Odd numbers are lo = ( 59 79 71 5 )
```

copy_n

Copies a specified number of elements.

C++

```
template<class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(
    InputIterator first,
    Size count,
    OutputIterator dest);

template<class ExecutionPolicy, class ForwardIterator1, class Size, class
ForwardIterator2>
ForwardIterator2 copy_n(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    Size count,
    ForwardIterator2 dest);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates where to copy elements from.

count

A signed or unsigned integer type specifying the number of elements to copy.

dest

An output iterator that indicates where to copy elements to.

Return value

Returns an output iterator where elements have been copied to. It's the same as the returned value of the *dest* parameter.

Remarks

The template function evaluates `*(dest + N) = *(first + N))` once for each `N` in the range `[0, count)`, for strictly increasing values of `N` starting with the lowest value. It then returns `dest + N`. If *`dest`* and *`first`* designate regions of storage, *`dest`* must not be in the range `[first, last)`.

Example

C++

```
// alg_copy_n.cpp
// compile with: cl /EHsc /W4 alg_copy_n.cpp
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    string s1{"dandelion"};
    string s2{"badger"};

    cout << s1 << " + " << s2 << " = ";

    // Copy the first 3 letters from s1
    // to the first 3 positions in s2
    copy_n(s1.begin(), 3, s2.begin());

    cout << s2 << endl;
}
```

Output

```
dandelion + badger = danger
```

count

Returns the number of elements in a range whose values match a specified value.

C++

```
template<class InputIterator, class Type>
typename iterator_traits<InputIterator>::difference_type count(
    InputIterator first,
    InputIterator last,
```

```
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class Type>
typename iterator_traits<ForwardIterator>::difference_type
count(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);
```

Parameters

exec

The execution policy to use.

first

An input iterator addressing the position of the first element in the range to be traversed.

last

An input iterator addressing the position one past the final element in the range to be traversed.

value

The value of the elements to be counted.

Return value

The difference type of the `InputIterator` that counts the number of elements in the range [*first*, *last*) that have value *value*.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

This algorithm is generalized to count elements that satisfy any predicate with the template function count_if.

Example

C++

```cpp
// alg_count.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result;
    result = count(v1.begin(), v1.end(), 10);
    cout << "The number of 10s in v2 is: " << result << "." << endl;
}
```

Output

```
v1 = ( 10 20 10 40 10 )
The number of 10s in v2 is: 3.
```

count_if

Returns the number of elements in a range whose values satisfy a specified condition.

C++

```cpp
template<class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type count_if(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate>
typename iterator_traits<ForwardIterator>::difference_type
count_if(
    ExecutionPolicy&& exec,
```

```
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec
The execution policy to use.

first
An input iterator addressing the position of the first element in the range to be searched.

last
An input iterator addressing the position one past the final element in the range to be searched.

pred
User-defined predicate function object that defines the condition to be satisfied if an element is to be counted. A unary predicate takes a single argument and returns **true** or **false**.

Return value

The number of elements that satisfy the condition specified by the predicate.

Remarks

This template function is a generalization of the algorithm count, replacing the predicate "equals a specific value" with any predicate.

Example

C++

```
// alg_count_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater10(int value)
{
```

```
    return value > 10;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(10);
    v1.push_back(40);
    v1.push_back(10);

    cout << "v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greater10);
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;
}
```

Output

```
v1 = ( 10 20 10 40 10 )
The number of elements in v1 greater than 10 is: 2.
```

`equal`

Compares two ranges element by element for equality or equivalence in a sense specified by a binary predicate.

Use `std::equal` when comparing elements in different container types (for example `vector` and `list`) or when comparing different element types, or when you need to compare subranges of containers. Otherwise, when comparing elements of the same type in the same container type, use the non-member `operator==` that is provided for each container.

Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range won't detect differences if the second range is longer than the first range. These overloads will result in undefined behavior if the second range is shorter than the first range.

C++

```
template<class InputIterator1, class InputIterator2>
bool equal(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    BinaryPredicate pred); // C++14

template<class InputIterator1, class InputIterator2>
bool equal(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
bool equal(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class BinaryPredicate>
bool equal(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
bool equal(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
```

```
    ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class BinaryPredicate>
bool equal(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first range to be tested.

last1

An input iterator addressing the position one past the last element in the first range to be tested.

first2

An input iterator addressing the position of the first element in the second range to be tested.

last2

An input iterator addressing the position of one past the last element in the second range to be tested.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **`true`** when satisfied and **`false`** when not satisfied.

Return value

`true` if and only if the ranges are identical or equivalent under the binary predicate when compared element by element; otherwise, **`false`**.

Remarks

The range to be searched must be valid; all iterators must be dereferenceable and the last position is reachable from the first by incrementation.

If the two ranges are equal length, then the time complexity of the algorithm is linear in the number of elements contained in the range. Otherwise the function immediately returns `false`.

You don't need either `operator==` or the user-defined predicate to impose an equivalence relation that's symmetric, reflexive, and transitive between its operands.

Example

C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v1 { 0, 5, 10, 15, 20, 25 };
    vector<int> v2 { 0, 5, 10, 15, 20, 25 };
    vector<int> v3 { 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };

    // Using range-and-a-half equal:
    bool b = equal(v1.begin(), v1.end(), v2.begin());
    cout << "v1 and v2 are equal: "
       << b << endl; // true, as expected

    b = equal(v1.begin(), v1.end(), v3.begin());
    cout << "v1 and v3 are equal: "
       << b << endl; // true, surprisingly

    // Using dual-range equal:
    b = equal(v1.begin(), v1.end(), v3.begin(), v3.end());
    cout << "v1 and v3 are equal with dual-range overload: "
       << b << endl; // false

    return 0;
}
```

Output

```
v1 and v2 are equal: 1
v1 and v3 are equal: 1
```

```
v1 and v3 are equal with dual-range overload: 0
```

equal_range

Given an ordered range, finds the subrange in which all elements are equivalent to a given value.

```
C++
```

```
template<class ForwardIterator, class Type>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ForwardIterator, class Type, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    Compare pred);
```

Parameters

first
A forward iterator addressing the position of the first element in the range to be searched.

last
A forward iterator addressing the position one past the final element in the range to be searched.

value
The value being searched for in the ordered range.

pred
User-defined predicate function object that defines the sense in which one element is less than another. A comparison predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

A pair of forward iterators that specify a subrange, contained within the range searched, in which all of the elements are equivalent to *value* in the sense defined by the binary predicate used (either *pred* or the default, less-than).

If no elements in the range are equivalent to *value*, the forward iterators in the returned pair are equal and specify the point where *value* could be inserted without disturbing the order of the range.

Remarks

The first iterator of the pair returned by the algorithm is lower_bound, and the second iterator is upper_bound.

The range must be sorted according to the predicate provided to `equal_range`. For example, if you're going to use the greater-than predicate, the range must be sorted in descending order.

Elements in the possibly empty subrange defined by the pair of iterators returned by `equal_range` will be equivalent to *value* in the sense defined by the predicate used.

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (*last* - *first*).

Example

C++

```
// alg_equal_range.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>()
#include <iostream>
#include <string>
using namespace std;

template<class T> void dump_vector( const vector<T>& v, pair<typename
vector<T>::iterator, typename vector<T>::iterator> range )
{
    // prints vector v with range delimited by [ and ]

    for ( typename vector<T>::const_iterator i = v.begin(); i != v.end();
++i )
    {
        if ( i == range.first )
        {
            cout << "[ ";
```

```
        }
        if ( i == range.second )
        {
            cout << "] ";
        }

        cout << *i << " ";
    }
    cout << endl;
}

template<class T> void equal_range_demo( const vector<T>& original_vector, T
value )
{
    vector<T> v(original_vector);

    sort( v.begin(), v.end() );
    cout << "Vector sorted by the default binary predicate <:" << endl <<
'\t';
    for ( typename vector<T>::const_iterator i = v.begin(); i != v.end();
++i )
    {
        cout << *i << " ";
    }
    cout << endl << endl;

    pair<typename vector<T>::iterator, typename vector<T>::iterator> result
        = equal_range( v.begin(), v.end(), value );

    cout << "Result of equal_range with value = " << value << ":" << endl <<
'\t';
    dump_vector( v, result );
    cout << endl;
}

template<class T, class F> void equal_range_demo( const vector<T>&
original_vector, T value, F pred, string predname )
{
    vector<T> v(original_vector);

    sort( v.begin(), v.end(), pred );
    cout << "Vector sorted by the binary predicate " << predname << ":" <<
endl << '\t';
    for ( typename vector<T>::const_iterator i = v.begin(); i != v.end();
++i )
    {
        cout << *i << " ";
    }
    cout << endl << endl;

    pair<typename vector<T>::iterator, typename vector<T>::iterator> result
        = equal_range( v.begin(), v.end(), value, pred );

    cout << "Result of equal_range with value = " << value << ":" << endl <<
'\t';
```

```
    dump_vector( v, result );
    cout << endl;
}

// Return whether absolute value of elem1 is less than absolute value of
elem2
bool abs_lesser( int elem1, int elem2 )
{
    return abs(elem1) < abs(elem2);
}

// Return whether string l is shorter than string r
bool shorter_than(const string& l, const string& r)
{
    return l.size() < r.size();
}

int main()
{
    vector<int> v1;

    // Constructing vector v1 with default less than ordering
    for ( int i = -1; i <= 4; ++i )
    {
        v1.push_back(i);
    }

    for ( int i =-3; i <= 0; ++i )
    {
        v1.push_back( i );
    }

    equal_range_demo( v1, 3 );
    equal_range_demo( v1, 3, greater<int>(), "greater" );
    equal_range_demo( v1, 3, abs_lesser, "abs_lesser" );

    vector<string> v2;

    v2.push_back("cute");
    v2.push_back("fluffy");
    v2.push_back("kittens");
    v2.push_back("fun");
    v2.push_back("meowmeowmeow");
    v2.push_back("blah");

    equal_range_demo<string>( v2, "fred" );
    equal_range_demo<string>( v2, "fred", shorter_than, "shorter_than" );
}
```

Output

```
Vector sorted by the default binary predicate <:
    -3 -2 -1 -1 0 0 1 2 3 4
```

```
Result of equal_range with value = 3:
    -3 -2 -1 -1 0 0 1 2 [ 3 ] 4

Vector sorted by the binary predicate greater:
    4 3 2 1 0 0 -1 -1 -2 -3

Result of equal_range with value = 3:
    4 [ 3 ] 2 1 0 0 -1 -1 -2 -3

Vector sorted by the binary predicate abs_lesser:
    0 0 -1 1 -1 2 -2 3 -3 4

Result of equal_range with value = 3:
    0 0 -1 1 -1 2 -2 [ 3 -3 ] 4

Vector sorted by the default binary predicate <:
    blah cute fluffy fun kittens meowmeowmeow

Result of equal_range with value = fred:
    blah cute fluffy [ ] fun kittens meowmeowmeow

Vector sorted by the binary predicate shorter_than:
    fun cute blah fluffy kittens meowmeowmeow

Result of equal_range with value = fred:
    fun [ cute blah ] fluffy kittens meowmeowmeow
```

`fill`

Assigns the same new value to every element in a specified range.

C++

```
template<class ForwardIterator, class Type>
void fill(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class Type>
void fill(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the range to be traversed.

last

A forward iterator addressing the position one past the final element in the range to be traversed.

value

The value to be assigned to elements in the range [*first*, *last*).

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

Example

C++

```
// alg_fill.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
```

```
    // Fill the last 5 positions with a value of 2
    fill( v1.begin( ) + 5, v1.end( ), 2 );

    cout << "Modified v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

`fill_n`

Assigns a new value to a specified number of elements in a range beginning with a particular element.

C++

```
template<class OutputIterator, class Size, class Type>
OutputIterator fill_n(
    OutputIterator first,
    Size count,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class Size, class
Type>
ForwardIterator fill_n(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    Size count,
    const Type& value);
```

Parameters

exec

The execution policy to use.

first

An output iterator addressing the position of the first element in the range to be assigned the value *value*.

count

A signed or unsigned integer type specifying the number of elements to be assigned the value.

value

The value to be assigned to elements in the range [*first*, *first + count*).

Return value

An iterator to the element that follows the last element filled if *count* > zero, otherwise the first element.

Remarks

The destination range must be valid; all pointers must be dereferenceable, and the last position is reachable from the first by incrementation. The complexity is linear with the size of the range.

Example

C++

```
// alg_fill_n.cpp
// compile using /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v;

    for ( auto i = 0 ; i < 9 ; ++i )
        v.push_back( 0 );

    cout << "vector v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;

    // Fill the first 3 positions with a value of 1, saving position.
    auto pos = fill_n( v.begin(), 3, 1 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
```

```
    cout << ")" << endl;

    // Fill the next 3 positions with a value of 2, using last position.
    fill_n( pos, 3, 2 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;

    // Fill the last 3 positions with a value of 3, using relative math.
    fill_n( v.end()-3, 3, 3 );

    cout << "modified v = ( " ;
    for ( const auto &w : v )
        cout << w << " ";
    cout << ")" << endl;
}
```

Output

```
vector v = ( 0 0 0 0 0 0 0 0 0 )
modified v = ( 1 1 1 0 0 0 0 0 0 )
modified v = ( 1 1 1 2 2 2 0 0 0 )
modified v = ( 1 1 1 2 2 2 3 3 3 )
```

find

Locates the position of the first occurrence of an element in a range that has a specified value.

C++

```
template<class InputIterator, class Type>
InputIterator find(
    InputIterator first,
    InputIterator last,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class Type>
ForwardIterator find(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);
```

Parameters

`exec`

The execution policy to use.

`first`

An input iterator addressing the position of the first element in the range to be searched for the specified value.

`last`

An input iterator addressing the position one past the final element in the range to be searched for the specified value.

`value`

The value to be searched for.

Return value

An input iterator addressing the first occurrence of the specified value in the range being searched. If no element is found with an equivalent value, returns `last`.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

For a code example using `find()`, see find_if.

`find_end`

Looks in a range for the last subsequence that is identical to a specified sequence or that is equivalent in a sense specified by a binary predicate.

C++

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class Pred>
ForwardIterator1 find_end(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
```

```
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    Pred pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator1
find_end(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_end(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);
```

Parameters

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the last element in the range to be searched.

first2

A forward iterator addressing the position of the first element in the range to search for.

last2

A forward iterator addressing the position one past the last element in the range to search for.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

A forward iterator addressing the position of the first element of the last subsequence within [first1, last1) that matches the specified sequence [first2, last2).

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

C++

```
// alg_find_end.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    list<int> L1;
    vector<int>::iterator Iter1, Iter2;
    list<int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
```

```
    for ( ii = 1 ; ii <= 4 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
        cout << ")" << endl;

    // Searching v1 for a match to L1 under identity
    vector<int>::iterator result1;
    result1 = find_end ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

    if ( result1 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a match of L1 in v1 that begins at "
            << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    result2 = find_end ( v1.begin( ), v1.end( ), v2.begin( ), v2.end( ),
twice );

    if ( result2 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a sequence of elements in v1 that "
            << "are equivalent to those\n in v2 under the binary "
            << "predicate twice and that begins at position "
            << result2 - v1.begin( ) << "." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 5 10 15 20 )
Vector v2 = ( 20 30 40 )
There is a match of L1 in v1 that begins at position 7.
There is a sequence of elements in v1 that are equivalent to those
in v2 under the binary predicate twice and that begins at position 8.
```

find_first_of

Searches for the first occurrence of any of several values within a target range. Or, searches for the first occurrence of any of several elements that are equivalent in a sense specified by a binary predicate to a specified set of the elements.

C++

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
ForwardIterator1 find_first_of(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator1
find_first_of(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_first_of(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);
```

Parameters

first1
A forward iterator addressing the position of the first element in the range to be searched.

last1
A forward iterator addressing the position one past the final element in the range to be searched.

first2
A forward iterator addressing the position of the first element in the range to be matched.

last2
A forward iterator addressing the position one past the final element in the range to be matched.

pred
User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The operator== used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position is reachable from the first by incrementation.

Example

```
C++
```

```cpp
// alg_find_first_of.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    list<int> L1;
    vector<int>::iterator Iter1, Iter2;
    list<int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
    for ( ii = 3 ; ii <= 4 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
```

```
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
        cout << ")" << endl;

    // Searching v1 for first match to L1 under identity
    vector<int>::iterator result1;
    result1 = find_first_of ( v1.begin( ), v1.end( ), L1.begin( ), L1.end( )
);

    if ( result1 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is at least one match of L1 in v1"
            << "\n and the first one begins at "
            << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    result2 = find_first_of ( v1.begin( ), v1.end( ), v2.begin( ), v2.end(
), twice );

    if ( result2 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is a sequence of elements in v1 that "
            << "are equivalent\n to those in v2 under the binary "
            << "predicate twice\n and the first one begins at position "
            << result2 - v1.begin( ) << "." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 15 20 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
and the first one begins at position 3.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.
```

find_if

Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

C++

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate>
ForwardIterator find_if(
    ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

first
An input iterator addressing the position of the first element in the range to be searched.

last
An input iterator addressing the position one past the final element in the range to be searched.

pred
User-defined predicate function object or lambda expression that defines the condition to be satisfied by the element being searched for. A unary predicate takes a single argument and returns **true** if satisfied, or **false** if not satisfied. The signature of *pred* must effectively be `bool pred(const T& arg);`, where `T` is a type to which `InputIterator` can be implicitly converted when dereferenced. The **const** keyword is shown only to illustrate that the function object or lambda shouldn't modify the argument.

Return value

An input iterator that refers to the first element in the range that satisfies the condition specified by the predicate (the predicate results in **true**). If no element is found to satisfy the predicate, returns *last*.

Remarks

This template function is a generalization of the algorithm find, replacing the predicate "equals a specific value" with any predicate. For the logical opposite (find the first element that doesn't satisfy the predicate), see find_if_not.

Example

C++

```
// cl.exe /W4 /nologo /EHsc /MTd
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

template <typename S> void print(const S& s) {
for (const auto& p : s) {
        cout << "(" << p << ") ";
    }
    cout << endl;
}

// Test std::find()
template <class InputIterator, class T>
void find_print_result(InputIterator first, InputIterator last, const T&
value) {

    // call <algorithm> std::find()
    auto p = find(first, last, value);

    cout << "value " << value;
    if (p == last) {
        cout << " not found." << endl;
    } else {
        cout << " found." << endl;
    }
}

// Test std::find_if()
template <class InputIterator, class Predicate>
void find_if_print_result(InputIterator first, InputIterator last,
    Predicate Pred, const string& Str) {

    // call <algorithm> std::find_if()
    auto p = find_if(first, last, Pred);

    if (p == last) {
        cout << Str << " not found." << endl;
    } else {
        cout << "first " << Str << " found: " << *p << endl;
    }
}

// Function to use as the UnaryPredicate argument to find_if() in this
example
bool is_odd_int(int i) {
    return ((i % 2) != 0);
```

```
}

int main()
{
    // Test using a plain old array.
    const int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    cout << "array x[] contents: ";
    print(x);
    // Using non-member std::begin()/std::end() to get input iterators for
the plain old array.
    cout << "Test std::find() with array..." << endl;
    find_print_result(begin(x), end(x), 10);
    find_print_result(begin(x), end(x), 42);
    cout << "Test std::find_if() with array..." << endl;
    find_if_print_result(begin(x), end(x), is_odd_int, "odd integer"); //
function name
    find_if_print_result(begin(x), end(x), // lambda
        [](int i){ return ((i % 2) == 0); }, "even integer");

    // Test using a vector.
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back((i + 1) * 10);
    }
    cout << endl << "vector v contents: ";
    print(v);
    cout << "Test std::find() with vector..." << endl;
    find_print_result(v.begin(), v.end(), 20);
    find_print_result(v.begin(), v.end(), 12);
    cout << "Test std::find_if() with vector..." << endl;
    find_if_print_result(v.begin(), v.end(), is_odd_int, "odd integer");
    find_if_print_result(v.begin(), v.end(), // lambda
        [](int i){ return ((i % 2) == 0); }, "even integer");
}
```

Output

```
array x[] contents: (1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Test std::find() with array...
value 10 found.
value 42 not found.
Test std::find_if() with array...
first odd integer found: 1
first even integer found: 2

vector v contents: (10) (20) (30) (40) (50) (60) (70) (80) (90) (100)
Test std::find() with vector...
value 20 found.
value 12 not found.
Test std::find_if() with vector...
odd integer not found.
first even integer found: 10
```

`find_if_not`

Returns the first element in the indicated range that doesn't satisfy a condition.

```
C++
```

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if_not(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate>
ForwardIterator find_if_not(
    ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

`first`
An input iterator addressing the position of the first element in the range to be searched.

`last`
An input iterator addressing the position one past the final element in the range to be searched.

`pred`
User-defined predicate function object or lambda expression that defines the condition to be not satisfied by the element being searched for. A unary predicate takes a single argument and returns **`true`** if satisfied, or **`false`** if not satisfied. The signature of *`pred`* must effectively be `bool pred(const T& arg);`, where `T` is a type to which `InputIterator` can be implicitly converted when dereferenced. The **`const`** keyword is shown only to illustrate that the function object or lambda shouldn't modify the argument.

Return value

An input iterator that refers to the first element in the range that doesn't satisfy the condition specified by the predicate (the predicate results in **`false`**). If all elements satisfy the predicate (the predicate results in **`true`** for every element), returns *`last`*.

Remarks

This template function is a generalization of the algorithm find, replacing the predicate "equals a specific value" with any predicate. For the logical opposite (find the first element that does satisfy the predicate), see find_if.

For a code example readily adaptable to `find_if_not()`, see find_if.

`for_each`

Applies a specified function object to each element in a forward order within a range and returns the function object.

C++

```
template<class InputIterator, class Function>
Function for_each(
    InputIterator first,
    InputIterator last,
    Function func);

template<class ExecutionPolicy, class ForwardIterator, class Function>
void for_each(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Function func);
```

Parameters

`first`
An input iterator addressing the position of the first element in the range to operate on.

`last`
An input iterator addressing the position one past the final element in the range operated on.

`func`
User-defined function object that is applied to each element in the range.

Return value

A copy of the function object after it has been applied to all of the elements in the range.

Remarks

The algorithm `for_each` is flexible, allowing the modification of each element within a range in different, user-specified ways. Templatized functions may be reused in a modified form by passing different parameters. User-defined functions may accumulate information within an internal state that the algorithm may return after processing all of the elements in the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear with at most (*`last`* - *`first`*) comparisons.

Example

C++

```
// alg_for_each.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
    Type Factor;   // The value to multiply by
public:
    // Constructor initializes the value to multiply by
    MultValue ( const Type& value ) : Factor ( value ) {
    }

    // The function call for the element to be multiplied
    void operator( ) ( Type& elem ) const
    {
        elem *= Factor;
    }
};

// The function object to determine the average
class Average
{
private:
    long num;      // The number of elements
    long sum;      // The sum of the elements
public:
    // Constructor initializes the value to multiply by
    Average( ) : num ( 0 ) , sum ( 0 )
```

```
    {
    }

    // The function call to process the next elment
    void operator( ) ( int elem )
    {
        num++;      // Increment the element count
        sum += elem;   // Add the value to the partial sum
    }

    // return Average
    operator double( )
    {
        return static_cast<double> (sum) /
            static_cast<double> (num);
    }
};

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    // Constructing vector v1
    int i;
    for ( i = -4 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Using for_each to multiply each element by a Factor
    for_each ( v1.begin( ), v1.end( ), MultValue<int> ( -2 ) );

    cout << "Multiplying the elements of the vector v1\n "
            << "by the factor -2 gives:\n v1mod1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // The function object is templatized and so can be
    // used again on the elements with a different Factor
    for_each ( v1.begin( ), v1.end( ), MultValue<int> ( 5 ) );

    cout << "Multiplying the elements of the vector v1mod\n "
            << "by the factor 5 gives:\n v1mod2 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
```

```
    // The local state of a function object can accumulate
    // information about a sequence of actions that the
    // return value can make available, here the Average
    double avemod2 = for_each ( v1.begin( ), v1.end( ),
        Average( ) );
    cout << "The average of the elements of v1 is:\n Average ( v1mod2 ) = "
         << avemod2 << "." << endl;
}
```

Output

```
Original vector v1 = ( -4 -3 -2 -1 0 1 2 ).
Multiplying the elements of the vector v1
by the factor -2 gives:
v1mod1 = ( 8 6 4 2 0 -2 -4 ).
Multiplying the elements of the vector v1mod
by the factor 5 gives:
v1mod2 = ( 40 30 20 10 0 -10 -20 ).
The average of the elements of v1 is:
Average ( v1mod2 ) = 10.
```

for_each_n

Applies a specified function object to a specified number of elements in a range beginning with a particular element.

C++

```
template<class InputIterator, class Size, class Function>
InputIterator for_each_n(
    InputIterator first,
    Size count,
    Function func);

template<class ExecutionPolicy, class ForwardIterator, class Size, class
Function>
ForwardIterator for_each_n(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    Size count,
    Function func);
```

Parameters

exec

The execution policy to use.

first

An input iterator at the position of the first element in the range to operate on.

count

The number of elements to operate on.

func

A user-defined function object to apply to each element in the range [`first`, `first` + `count`).

Return value

An iterator to the element that follows the last element processed if *count* > zero, otherwise the first element.

Remarks

count must be non-negative, and there must be at least *count* elements in the range starting from *first*.

Example

This example defines a function object class. Production code often uses a lambda to achieve the same result with less code.

C++

```
// alg_for_each_n.cpp
// compile with /EHsc and /std:c++17 (or higher)
#include <algorithm>
#include <iostream>
#include <vector>

// The function object multiplies an element by a Factor
template <class Type> class MultValue
{
private:
    Type Factor; // The value to multiply each element by
public:
    // Constructor initializes the value to multiply by
    MultValue(const Type &value) : Factor(value) {}

    // The function call for the element to be multiplied
    void operator()(Type &elem) const
    {
        elem *= Factor;
```

```
    }
};

// Utility to display the contents of a vector
template <class T> void print_vector(const std::vector<T> &vec)
{
    std::cout << "( ";

    for (auto iter = vec.begin(); iter != vec.end(); iter++)
    {
        std::cout << *iter << ' ';
    }

    std::cout << ").\n";
}

int main()
{
    std::vector<int> v;

    // Construct vector with the elements -4...2
    for (int i = -4; i <= 2; i++)
    {
        v.push_back(i);
    }

    std::cout << "Original vector v = ";
    print_vector(v);

    // Use for_each_n to multiply the first 3 elements by a Factor,
    // saving the position in the vector after the first 3 elements
    auto pos = for_each_n(v.begin(), 3, MultValue<int>(-2));

    std::cout << "Multiplying the first 3 elements of the vector v\n "
              << "by the factor -2 gives:\n vmod1 = ";
    print_vector(v);

    // Using for_each_n to multiply the next 4 elements by a Factor,
    // starting at the position saved by the previous for_each_n
    for_each_n(pos, 4, MultValue<int>(-3));

    std::cout << "Multiplying the next 4 elements of the vector v\n "
         << "by the factor -3 gives:\n vmod2 = ";
    print_vector(v);

    return 0;
}
```

Output

```
Original vector v = ( -4 -3 -2 -1 0 1 2 ).
Multiplying the first 3 elements of the vector v
 by the factor -2 gives:
```

```
 vmod1 = ( 8 6 4 -1 0 1 2 ).
Multiplying the next 4 elements of the vector v
 by the factor -3 gives:
 vmod2 = ( 8 6 4 3 0 -3 -6 ).
```

generate

Assigns the values generated by a function object to each element in a range.

C++

```
template<class ForwardIterator, class Generator>
void generate(
    ForwardIterator first,
    ForwardIterator last,
    Generator gen);

template<class ExecutionPolicy, class ForwardIterator, class Generator>
void generate(
    ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Generator gen);
```

Parameters

first

A forward iterator at the position of the first element in the range to which values are to be assigned.

last

A forward iterator at the position one past the final element in the range to which values are to be assigned.

gen

A function object that is called with no arguments to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and doesn't need to return the same value each time it's called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterator for the range.

The range referenced must be valid. All pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly `last - first` calls made to the generator.

Example

C++

```
// alg_generate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

int main()
{
    using namespace std;

    // Assigning random values to vector integer elements
    vector<int> v1 ( 5 );
    vector<int>::iterator Iter1;
    deque<int> deq1 ( 5 );
    deque<int>::iterator d1_Iter;

    generate ( v1.begin( ), v1.end( ), rand );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Assigning random values to deque integer elements
    generate ( deq1.begin( ), deq1.end( ), rand );

    cout << "Deque deq1 is ( " ;
    for ( d1_Iter = deq1.begin( ) ; d1_Iter != deq1.end( ) ; d1_Iter++ )
        cout << *d1_Iter << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 41 18467 6334 26500 19169 ).
Deque deq1 is ( 15724 11478 29358 26962 24464 ).
```

`generate_n`

Assigns the values generated by a function object to a specified number of elements in a range. Returns the position one past the last assigned value.

```
C++
```

```
template<class OutputIterator, class Diff, class Generator>
void generate_n(
    OutputIterator first,
    Diff count,
    Generator gen);

template<class ExecutionPolicy, class ForwardIterator, class Size, class
Generator>
ForwardIterator generate_n(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    Size count,
    Generator gen);
```

Parameters

exec

The execution policy to use.

first

An output iterator addressing the position of first element in the range to which values are to be assigned.

count

A signed or unsigned integer type specifying the number of elements to be assigned a value by the generator function.

gen

A function object that is called with no arguments that is used to generate the values to be assigned to each of the elements in the range.

Remarks

The function object is invoked for each element in the range and doesn't need to return the same value each time it's called. It may, for example, read from a file or refer to and modify a local state. The generator's result type must be convertible to the value type of the forward iterators for the range.

The range referenced must be valid; all pointers must be dereferenceable and, within the sequence, the last position must be reachable from the first by incrementation.

The complexity is linear, with exactly `count` calls to the generator being required.

Example

C++

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <vector>
#include <deque>
#include <iostream>
#include <string>
#include <algorithm>
#include <random>

using namespace std;

template <typename C>
void print(const string& s, const C& c)
{
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    const int elemcount = 5;
    vector<int> v(elemcount);
    deque<int> dq(elemcount);

    // Set up random number distribution
    random_device rd;
    mt19937 engine(rd());
    uniform_int_distribution<int> dist(-9, 9);

    // Call generate_n, using a lambda for the third parameter
    generate_n(v.begin(), elemcount, [&](){ return dist(engine); });
    print("vector v is: ", v);

    generate_n(dq.begin(), elemcount, [&](){ return dist(engine); });
    print("deque dq is: ", dq);
}
```

Output

```
vector v is: 5 8 2 -9 6
deque dq is: 7 6 9 3 4
```

`includes`

Tests whether one sorted range contains all the elements contained in a second sorted range, where the ordering or equivalence criterion between elements may be specified by a binary predicate.

C++

```
template<class InputIterator1, class InputIterator2>
bool includes(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
bool includes(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Compare>
bool includes(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    Compare pred);
```

Parameters

exec
The execution policy to use.

first1
An input iterator addressing the position of the first element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

last1
An input iterator addressing the position one past the last element in the first of two sorted source ranges to be tested for whether all the elements of the second are contained in the first.

first2
An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

last2
An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be tested for whether all the elements of the second are contained in the first.

pred
User-defined predicate function object that defines sense in which one element is less than another. A comparison predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

true if the first sorted range contains all the elements in the second sorted range; otherwise, **false**.

Remarks

Another way to think of this test is that it determined whether the second source range is a subset of the first source range.

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

As a precondition to the application of the algorithm `includes`, the sorted source ranges must each be arranged with the same ordering that's used by the algorithm to sort the combined ranges.

The source ranges aren't modified by the algorithm `merge`.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. More precisely, the algorithm tests whether all the elements in the first sorted range under a specified binary predicate have equivalent ordering to the ones in the second sorted range.

The complexity of the algorithm is linear with at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons for nonempty source ranges.

Example

C++

```
// alg_includes.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1a, v1b;
    vector<int>::iterator Iter1a, Iter1b;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -2 ; i <= 4 ; i++ )
    {
        v1a.push_back( i );
    }
```

```
int ii;
for ( ii =-2 ; ii <= 3 ; ii++ )
{
    v1b.push_back( ii );
}

cout << "Original vector v1a with range sorted by the\n "
        << "binary predicate less than is v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
        << "binary predicate less than is v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector<int> v2a ( v1a ) , v2b ( v1b );
vector<int>::iterator Iter2a, Iter2b;
sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );
v2a.pop_back( );

cout << "Original vector v2a with range sorted by the\n "
        << "binary predicate greater is v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
        << "binary predicate greater is v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector<int> v3a ( v1a ), v3b ( v1b ) ;
vector<int>::iterator Iter3a, Iter3b;
reverse (v3a.begin( ), v3a.end( ) );
v3a.pop_back( );
v3a.pop_back( );
sort ( v3a.begin( ), v3a.end( ), mod_lesser );
sort ( v3b.begin( ), v3b.end( ), mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
        << "binary predicate mod_lesser is v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;

cout << "Original vector v3b with range sorted by the\n "
        << "binary predicate mod_lesser is v3b = ( " ;
```

```
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To test for inclusion under an asscending order
    // with the default binary predicate less<int>( )
    bool Result1;
    Result1 = includes ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ) );
    if ( Result1 )
        cout << "All the elements in vector v1b are "
            << "contained in vector v1a." << endl;
    else
        cout << "At least one of the elements in vector v1b "
            << "is not contained in vector v1a." << endl;

    // To test for inclusion under descending
    // order specify binary predicate greater<int>( )
    bool Result2;
    Result2 = includes ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ), greater<int>( ) );
    if ( Result2 )
        cout << "All the elements in vector v2b are "
            << "contained in vector v2a." << endl;
    else
        cout << "At least one of the elements in vector v2b "
            << "is not contained in vector v2a." << endl;

    // To test for inclusion under a user
    // defined binary predicate mod_lesser
    bool Result3;
    Result3 = includes ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), mod_lesser );
    if ( Result3 )
        cout << "All the elements in vector v3b are "
            << "contained under mod_lesser in vector v3a."
            << endl;
    else
        cout << "At least one of the elements in vector v3b is "
            << " not contained under mod_lesser in vector v3a."
            << endl;
}
```

Output

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -2 -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -2 -1 0 1 2 3 ).
Original vector v2a with range sorted by the
binary predicate greater is v2a = ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
binary predicate greater is v2b = ( 3 2 1 0 -1 -2 ).
```

```
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 1 2 3 4 ).
Original vector v3b with range sorted by the
binary predicate mod_lesser is v3b = ( 0 -1 1 -2 2 3 ).
All the elements in vector v1b are contained in vector v1a.
At least one of the elements in vector v2b is not contained in vector v2a.
At least one of the elements in vector v3b is not contained under mod_lesser
in vector v3a.
```

`inplace_merge`

Combines the elements from two consecutive sorted ranges into a single sorted range, where the ordering criterion may be specified by a binary predicate.

C++

```
template<class BidirectionalIterator>
void inplace_merge(
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last,
    Compare pred);

template<class ExecutionPolicy, class BidirectionalIterator>
void inplace_merge(
    ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
void inplace_merge(
    ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first
A bidirectional iterator addressing the position of the first element in the first of two consecutive sorted ranges to be combined and sorted into a single range.

middle
A bidirectional iterator addressing the position of the first element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

last
A bidirectional iterator addressing the position one past the last element in the second of two consecutive sorted ranges to be combined and sorted into a single range.

pred
User-defined predicate function object that defines the sense in which one element is less than another. The comparison predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Remarks

The sorted consecutive ranges referenced must be valid; all pointers must be dereferenceable and, within each sequence, the last position must be reachable from the first by incrementation.

The sorted consecutive ranges must each be arranged as a precondition to the application of the `inplace_merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges. The operation is stable as the relative order of elements within each range is preserved. When there are equivalent elements in both source ranges, the element is the first range precedes the element from the second in the combined range.

The complexity depends on the available memory as the algorithm allocates memory to a temporary buffer. If sufficient memory is available, the best case is linear with `(last - first) - 1` comparisons; if no auxiliary memory is available, the worst case is `N log(N)`, where *N* = *last* - *first*.

Example

C++

```
// alg_inplace_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
```

```
#include <functional>      //For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1, Iter2, Iter3;

    // Constructing vector v1 with default less-than ordering
    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii =-5 ; ii <= 0 ; ii++ )
    {
        v1.push_back( ii );
    }

    cout << "Original vector v1 with subranges sorted by the\n "
            << "binary predicate less than is v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Constructing vector v2 with ranges sorted by greater
    vector<int> v2 ( v1 );
    vector<int>::iterator break2;
    break2 = find ( v2.begin( ), v2.end( ), -5 );
    sort ( v2.begin( ), break2 , greater<int>( ) );
    sort ( break2 , v2.end( ), greater<int>( ) );
    cout << "Original vector v2 with subranges sorted by the\n "
            << "binary predicate greater is v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Constructing vector v3 with ranges sorted by mod_lesser
    vector<int> v3 ( v1 );
    vector<int>::iterator break3;
    break3 = find ( v3.begin( ), v3.end( ), -5 );
    sort ( v3.begin( ), break3 , mod_lesser );
```

```
    sort ( break3 , v3.end( ), mod_lesser );
    cout << "Original vector v3 with subranges sorted by the\n "
            << "binary predicate mod_lesser is v3 = ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;

    vector<int>::iterator break1;
    break1 = find (v1.begin( ), v1.end( ), -5 );
    inplace_merge ( v1.begin( ), break1, v1.end( ) );
    cout << "Merged inplace with default order,\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To merge inplace in descending order, specify binary
    // predicate greater<int>( )
    inplace_merge ( v2.begin( ), break2 , v2.end( ) , greater<int>( ) );
    cout << "Merged inplace with binary predicate greater specified,\n "
            << "vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")" << endl;

    // Applying a user defined (UD) binary predicate mod_lesser
    inplace_merge ( v3.begin( ), break3, v3.end( ), mod_lesser );
    cout << "Merged inplace with binary predicate mod_lesser specified,\n "
            << "vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")" << endl;
}
```

Output

```
Original vector v1 with subranges sorted by the
binary predicate less than is v1 = ( 0 1 2 3 4 5 -5 -4 -3 -2 -1 0 )
Original vector v2 with subranges sorted by the
binary predicate greater is v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Original vector v3 with subranges sorted by the
binary predicate mod_lesser is v3 = ( 0 1 2 3 4 5 0 -1 -2 -3 -4 -5 )
Merged inplace with default order,
vector v1mod = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 )
Merged inplace with binary predicate greater specified,
vector v2mod = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Merged inplace with binary predicate mod_lesser specified,
vector v3mod = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 )
```

is_heap

Returns `true` if the elements in the specified range form a heap.

C++

```
template<class RandomAccessIterator>
bool is_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);

template<class ExecutionPolicy, class RandomAccessIterator>
bool is_heap(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
bool is_heap(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A random access iterator that indicates the start of a range to check for a heap.

last

A random access iterator that indicates the end of a range.

pred

A condition to test to order elements. A comparison predicate takes two arguments and returns `true` or `false`.

Return value

Returns `true` if the elements in the specified range form a heap, `false` if they don't.

Remarks

The first template function returns is_heap_until `(first , last) == last`.

The second template function returns

`is_heap_until(first, last, pred) == last`.

is_heap_until

Returns an iterator positioned at the first element in the range [`first`, `last`) that doesn't satisfy the heap ordering condition, or *`end`* if the range forms a heap.

C++

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);

template<class ExecutionPolicy, class RandomAccessIterator>
RandomAccessIterator is_heap_until(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

`exec`

The execution policy to use.

`first`

A random access iterator that specifies the first element of a range to check for a heap.

last

A random access iterator that specifies the end of the range to check for a heap.

pred

A binary predicate that specifies the strict weak ordering condition that defines a heap. The default predicate is `std::less<>` when *pred* isn't specified.

Return value

Returns *last* if the specified range forms a heap or contains one or fewer elements. Otherwise, returns an iterator for the first element found that doesn't satisfy the heap condition.

Remarks

The first template function returns the last iterator `next` in `[first, last)` where `[first, next)` is a heap ordered by the function object `std::less<>`. If the distance `last - first` is less than 2, the function returns *last*.

The second template function behaves the same as the first, except that it uses the predicate *pred* instead of `std::less<>` as the heap ordering condition.

is_partitioned

Returns **`true`** if all the elements in the given range that test **`true`** for a condition come before any elements that test **`false`**.

C++

```
template<class InputIterator, class UnaryPredicate>
bool is_partitioned(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template <class ExecutionPolicy, class ForwardIterator, class
UnaryPredicate>
bool is_partitioned(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates where a range starts to check for a condition.

last

An input iterator that indicates the end of a range.

pred

The condition to test for. This test is provided by a user-defined predicate function object that defines the condition to be satisfied by the element being searched for. A unary predicate takes a single argument and returns **true** or **false**.

Return value

Returns **true** when all of the elements in the given range that test **true** for a condition come before any elements that test **false**, and otherwise, returns **false**.

Remarks

The template function returns **true** only if all elements in `[first, last)` are partitioned by *pred*; that is, all elements `X` in `[first, last)` for which `pred (X)` is true occur before all elements `Y` for which `pred (Y)` is **false**.

is_permutation

Returns `true` if both ranges contain the same elements, whether or not the elements are in the same order. Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range won't detect differences if the second range is longer than the first range. These overloads will result in undefined behavior if the second range is shorter than the first range.

C++

```
template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2);
```

```
template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
bool is_permutation(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    BinaryPredicate Pred);

// C++14
template<class ForwardIterator1, class ForwardIterator2>
bool is_permutation(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
bool is_permutation(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);
```

Parameters

first1

A forward iterator that refers to the first element of the range.

last1

A forward iterator that refers one past the last element of the range.

first2

A forward iterator that refers to the first element of a second range, used for comparison.

last2

A forward iterator that refers to one past the last element of a second range, used for comparison.

pred

A predicate that tests for equivalence and returns a **bool**.

Return value

`true` when the ranges can be rearranged so as to be identical according to the comparator predicate; otherwise, `false`.

Remarks

`is_permutation` has quadratic complexity in the worst case.

The first template function assumes that there are as many elements in the range beginning at *`first2`* as there are in the range designated by `[first1, last1)`. If there are more elements in the second range, they're ignored; if there are less, undefined behavior will occur. The third template function (C++14 and later) doesn't make this assumption. Both return `true` only if, for each element `X` in the range designated by `[first1, last1)` there are as many elements `Y` in the same range for which `X == Y` as there are in the range beginning at *`first2`* or `[first2, last2)`. Here, `operator==` must perform a pairwise comparison between its operands.

The second and fourth template functions behave the same, except that they replace `operator==(X, Y)` with `Pred(X, Y)`. To behave correctly, the predicate must be symmetric, reflexive, and transitive.

Example

The following example shows how to use `is_permutation`:

C++

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;

int main()
{
    vector<int> vec_1{ 2, 3, 0, 1, 4, 5 };
    vector<int> vec_2{ 5, 4, 0, 3, 1, 2 };

    vector<int> vec_3{ 4, 9, 13, 3, 6, 5 };
    vector<int> vec_4{ 7, 4, 11, 9, 2, 1 };

    cout << "(1) Compare using built-in == operator: ";
    cout << boolalpha << is_permutation(vec_1.begin(), vec_1.end(),
        vec_2.begin(), vec_2.end()) << endl; // true

    cout << "(2) Compare after modifying vec_2: ";
```

```
    vec_2[0] = 6;
    cout << is_permutation(vec_1.begin(), vec_1.end(),
        vec_2.begin(), vec_2.end()) << endl; // false

    // Define equivalence as "both are odd or both are even"
    cout << "(3) vec_3 is a permutation of vec_4: ";
    cout << is_permutation(vec_3.begin(), vec_3.end(),
        vec_4.begin(), vec_4.end(),
        [](int lhs, int rhs) { return lhs % 2 == rhs % 2; }) << endl; //
true

    // Initialize a vector using the 's' string literal to specify a
std::string
    vector<string> animals_1{ "dog"s, "cat"s, "bird"s, "monkey"s };
    vector<string> animals_2{ "donkey"s, "bird"s, "meerkat"s, "cat"s };

    // Define equivalence as "first letters are equal":
    bool is_perm = is_permutation(animals_1.begin(), animals_1.end(),
animals_2.begin(), animals_2.end(),
        [](const string& lhs, const string& rhs)
    {
        return lhs[0] == rhs[0]; //std::string guaranteed to have at least a
null terminator
    });

    cout << "animals_2 is a permutation of animals_1: " << is_perm << endl;
// true

    return 0;
}
```

Output

```
(1) Compare using built-in == operator: true
(2) Compare after modifying vec_2: false
(3) vec_3 is a permutation of vec_4: true
animals_2 is a permutation of animals_1: true
```

is_sorted

Returns `true` if the elements in the specified range are in sorted order.

C++

```
template<class ForwardIterator>
bool is_sorted(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class Compare>
```

```
bool is_sorted(
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);

template<class ExecutionPolicy, class ForwardIterator>
bool is_sorted(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
bool is_sorted(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator that indicates where the range to check begins.

last

A forward iterator that indicates the end of a range.

pred

The condition to test to determine an order between two elements. A comparison predicate takes two arguments and returns **true** or **false**. This predicate performs the same task as `operator<`.

Remarks

The first template function returns is_sorted_until`( first, last ) == last`. The `operator<` function performs the order comparison.

The second template function returns `is_sorted_until( first, last , pred ) == last`. The *pred* predicate function performs the order comparison.

is_sorted_until

Returns a `ForwardIterator` that is set to the last element that is in sorted order from a specified range.

The second version lets you provide a comparison function object that returns **`true`** when two given elements are in sorted order, and **`false`** otherwise.

C++

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator is_sorted_until(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

Parameters

`exec`

The execution policy to use.

`first`

A forward iterator that indicates where the range to check starts.

`last`

A forward iterator that indicates the end of a range.

`pred`

The condition to test to determine an order between two elements. A comparison predicate takes two arguments and returns **`true`** or **`false`**.

Return value

Returns a `ForwardIterator` set to the last element in sorted order. The sorted sequence starts from *`first`*.

Remarks

The first template function returns the last iterator `next` in `[first, last]` so that `[first, next)` is a sorted sequence ordered by `operator<`. If `distance()` is less than 2, the function returns *`last`*.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pred(X, Y)`.

`iter_swap`

Exchanges two values referred to by a pair of specified iterators.

```
C++

template<class ForwardIterator1, class ForwardIterator2>
void iter_swap( ForwardIterator1 left, ForwardIterator2 right );
```

Parameters

`left`
One of the forward iterators whose value is to be exchanged.

`right`
The second of the forward iterators whose value is to be exchanged.

Remarks

`swap` should be used in preference to **iter_swap**, which was included in the C++ Standard for backward compatibility. If `Fit1` and `Fit2` are forward iterators, then `iter_swap( Fit1, Fit2 )`, is equivalent to `swap( *Fit1, *Fit2 )`.

The value types of the input forward iterators must have the same value.

Example

C++

```
// alg_iter_swap.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt&   operator=( const CInt& rhs ) { m_nVal =
    rhs.m_nVal; return *this; }
    bool operator<( const CInt& rhs ) const
        { return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt(" << rhs.m_nVal << ")";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    CInt c1 = 5, c2 = 1, c3 = 10;
    deque<CInt> deq1;
    deque<CInt>::iterator d1_Iter;

    deq1.push_back ( c1 );
    deq1.push_back ( c2 );
    deq1.push_back ( c3 );
```

```
    cout << "The original deque of CInts is deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Exchanging first and last elements with iter_swap
    iter_swap ( deq1.begin( ), --deq1.end( ) );

    cout << "The deque of CInts with first & last elements swapped is:\n
deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Swapping back first and last elements with swap
    swap ( *deq1.begin( ), *(deq1.end( ) -1 ) );

    cout << "The deque of CInts with first & last elements swapped back
is:\n deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    // Swapping a vector element with a deque element
    vector<int> v1;
    vector<int>::iterator Iter1;
    deque<int> deq2;
    deque<int>::iterator d2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 4 ; ii <= 5 ; ii++ )
    {
        deq2.push_back( ii );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Deque deq2 is ( " ;
    for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
        cout << *d2_Iter << " ";
    cout << ")." << endl;
```

```
    iter_swap ( v1.begin( ), deq2.begin( ) );

    cout << "After exchanging first elements,\n vector v1 is: v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl << " & deque deq2 is: deq2 = ( ";
    for ( d2_Iter = deq2.begin( ) ; d2_Iter != deq2.end( ) ; d2_Iter++ )
        cout << *d2_Iter << " ";
    cout << ")." << endl;
}
```

Output

```
The original deque of CInts is deq1 = ( CInt(5), CInt(1), CInt(10) ).
The deque of CInts with first & last elements swapped is:
deq1 = ( CInt(10), CInt(1), CInt(5) ).
The deque of CInts with first & last elements swapped back is:
deq1 = ( CInt(5), CInt(1), CInt(10) ).
Vector v1 is ( 0 1 2 3 ).
Deque deq2 is ( 4 5 ).
After exchanging first elements,
vector v1 is: v1 = ( 4 1 2 3 ).
& deque deq2 is: deq2 = ( 0 5 ).
```

lexicographical_compare

Compares element by element between two sequences to determine which is lesser of the two.

C++

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2 );

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
bool lexicographical_compare(
    ExecutionPolicy&& exec,
```

```
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Compare>
bool lexicographical_compare(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first range to be compared.

last1

An input iterator addressing the position one past the final element in the first range to be compared.

first2

An input iterator addressing the position of the first element in the second range to be compared.

last2

An input iterator addressing the position one past the final element in the second range to be compared.

pred

User-defined predicate function object that defines sense in which one element is less than another. A comparison predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

true if the first range is lexicographically less than the second range; otherwise **false**.

Remarks

A lexicographical comparison between sequences compares them element by element until:

- It finds two corresponding elements unequal, and the result of their comparison is taken as the result of the comparison between sequences.
- No inequalities are found, but one sequence has more elements than the other, and the shorter sequence is considered less than the longer sequence.
- No inequalities are found and the sequences have the same number of elements, and so the sequences are equal and the result of the comparison is `false`.

Example

C++

```
// alg_lex_comp.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    list<int> L1;
    vector<int>::iterator Iter1, Iter2;
    list<int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }
    int ii;
    for ( ii = 0 ; ii <= 6 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }
```

```
    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
        cout << ")" << endl;

    // Self lexicographical_comparison of v1 under identity
    bool result1;
    result1 = lexicographical_compare (v1.begin( ), v1.end( ),
                    v1.begin( ), v1.end( ) );
    if ( result1 )
        cout << "Vector v1 is lexicographically_less than v1." << endl;
    else
        cout << "Vector v1 is not lexicographically_less than v1." << endl;

    // lexicographical_comparison of v1 and L2 under identity
    bool result2;
    result2 = lexicographical_compare (v1.begin( ), v1.end( ),
                    L1.begin( ), L1.end( ) );
    if ( result2 )
        cout << "Vector v1 is lexicographically_less than L1." << endl;
    else
        cout << "Vector v1 is lexicographically_less than L1." << endl;

    bool result3;
    result3 = lexicographical_compare (v1.begin( ), v1.end( ),
                    v2.begin( ), v2.end( ), twice );
    if ( result3 )
        cout << "Vector v1 is lexicographically_less than v2 "
            << "under twice." << endl;
    else
        cout << "Vector v1 is not lexicographically_less than v2 "
            << "under twice." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 )
List L1 = ( 0 5 10 15 20 25 30 )
Vector v2 = ( 0 10 20 30 40 50 )
Vector v1 is not lexicographically_less than v1.
Vector v1 is lexicographically_less than L1.
Vector v1 is not lexicographically_less than v2 under twice.
```

`lower_bound`

Finds the position of the first element in an ordered range that has a value greater than or equivalent to a specified value. The ordering criterion may be specified by a binary predicate.

C++

```
template<class ForwardIterator, class Type>
ForwardIterator lower_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value );

template<class ForwardIterator, class Type, class BinaryPredicate>
ForwardIterator lower_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    BinaryPredicate pred );
```

Parameters

`first`

A forward iterator addressing the position of the first element in the range to be searched.

`last`

A forward iterator addressing the position one past the final element in the range to be searched.

`value`

The value whose first position or possible first position is being searched for in the ordered range.

`pred`

User-defined predicate function object that defines sense in which one element is less

than another. A binary predicate takes two arguments and returns `true` when satisfied and `false` when not satisfied.

Return value

A forward iterator at the position of the first element in an ordered range with a value that is greater than or equivalent to a specified value. The equivalence may be specified with a binary predicate.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of using `lower_bound` and where the ordering is the same as specified by with binary predicate.

The range isn't modified by the algorithm `lower_bound`.

The value types of the forward iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements.

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (`last - first`).

Example

C++

```
// alg_lower_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
```

```
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    vector<int> v1;
    // Constructing vector v1 with default less-than ordering
    for ( auto i = -1 ; i <= 4 ; ++i )
    {
        v1.push_back( i );
    }

    for ( auto ii =-3 ; ii <= 0 ; ++ii )
    {
        v1.push_back( ii );
    }

    cout << "Starting vector v1 = ( " ;
    for (const auto &Iter : v1)
        cout << Iter << " ";
    cout << ")." << endl;

    sort(v1.begin(), v1.end());
    cout << "Original vector v1 with range sorted by the\n "
        << "binary predicate less than is v1 = ( " ;
    for (const auto &Iter : v1)
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vector v2 with range sorted by greater
    vector<int> v2(v1);

    sort(v2.begin(), v2.end(), greater<int>());

    cout << "Original vector v2 with range sorted by the\n "
        << "binary predicate greater is v2 = ( " ;
    for (const auto &Iter : v2)
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vectors v3 with range sorted by mod_lesser
    vector<int> v3(v1);
    sort(v3.begin(), v3.end(), mod_lesser);

    cout << "Original vector v3 with range sorted by the\n "
        << "binary predicate mod_lesser is v3 = ( " ;
    for (const auto &Iter : v3)
        cout << Iter << " ";
    cout << ")." << endl;

    // Demonstrate lower_bound
```

```
    vector<int>::iterator Result;

    // lower_bound of 3 in v1 with default binary predicate less<int>()
    Result = lower_bound(v1.begin(), v1.end(), 3);
    cout << "The lower_bound in v1 for the element with a value of 3 is: "
         << *Result << "." << endl;

    // lower_bound of 3 in v2 with the binary predicate greater<int>( )
    Result = lower_bound(v2.begin(), v2.end(), 3, greater<int>());
    cout << "The lower_bound in v2 for the element with a value of 3 is: "
         << *Result << "." << endl;

    // lower_bound of 3 in v3 with the binary predicate mod_lesser
    Result = lower_bound(v3.begin(), v3.end(), 3, mod_lesser);
    cout << "The lower_bound in v3 for the element with a value of 3 is: "
         << *Result << "." << endl;
}
```

Output

```
Starting vector v1 = ( -1 0 1 2 3 4 -3 -2 -1 0 ).
Original vector v1 with range sorted by the
 binary predicate less than is v1 = ( -3 -2 -1 -1 0 0 1 2 3 4 ).
Original vector v2 with range sorted by the
 binary predicate greater is v2 = ( 4 3 2 1 0 0 -1 -1 -2 -3 ).
Original vector v3 with range sorted by the
 binary predicate mod_lesser is v3 = ( 0 0 -1 -1 1 -2 2 -3 3 4 ).
The lower_bound in v1 for the element with a value of 3 is: 3.
The lower_bound in v2 for the element with a value of 3 is: 3.
The lower_bound in v3 for the element with a value of 3 is: -3.
```

make_heap

Converts elements from a specified range into a heap in which the first element is the largest and for which a sorting criterion may be specified with a binary predicate.

C++

```
template<class RandomAccessIterator>
void make_heap(
    RandomAccessIterator first,
    RandomAccessIterator last );

template<class RandomAccessIterator, class BinaryPredicate>
void make_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    BinaryPredicate pred );
```

Parameters

first
A random-access iterator addressing the position of the first element in the range to be converted into a heap.

last
A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

pred
User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they're used in the implementation of the C++ Standard Library container adaptor priority_queue Class.

The complexity is linear, requiring `3 * (last - first)` comparisons.

Example

C++

```
// alg_make_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
```

```
        v1.push_back( i );

    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with default less than ordering
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with greater than ordering
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater-than heaped version of v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 4 3 7 8 0 5 2 1 6 9 ).
The heaped version of vector v1 is ( 9 8 7 6 3 5 2 1 4 0 ).
The greater-than heaped version of v1 is ( 0 1 2 4 3 5 7 6 9 8 ).
```

max

Compares two objects and returns the larger of the two, where the ordering criterion may be specified by a binary predicate.

C++

```
template<class Type>
constexpr Type& max(
    const Type& left,
    const Type& right);
template<class Type, class Pr>
constexpr Type& max(
    const Type& left,
    const Type& right,
    BinaryPredicate pred);
template<class Type>
constexpr Type& max (
    initializer_list<Type> ilist);
template<class Type, class Pr>
```

```
constexpr Type& max(
    initializer_list<Type> ilist,
    BinaryPredicate pred);
```

Parameters

left

The first of the two objects being compared.

right

The second of the two objects being compared.

pred

A binary predicate used to compare the two objects.

inlist

The initializer list that contains the objects to be compared.

Return value

The greater of the two objects, unless neither is greater; in that case, it returns the first of the two objects. When an `initializer_list` is specified, it returns the greatest of the objects in the list.

Remarks

The `max` algorithm is unusual in having objects passed as parameters. Most C++ Standard Library algorithms operate on a range of elements whose position is specified by iterators passed as parameters. If you need a function that operates on a range of elements, use max_element instead. Visual Studio 2017 enables **`constexpr`** on the overloads that take an `initializer_list`.

Example

C++

```
// alg_max.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>
```

```
using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt&   operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether absolute value of elem1 is greater than
// absolute value of elem2
bool abs_greater ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = -elem1;
    if ( elem2 < 0 )
        elem2 = -elem2;
    return elem1 < elem2;
};

int main()
{
    int a = 6, b = -7;
    // Return the integer with the larger absolute value
    const int& result1 = max(a, b, abs_greater);
    // Return the larger integer
    const int& result2 = max(a, b);

    cout << "Using integers 6 and -7..." << endl;
    cout << "The integer with the greater absolute value is: "
            << result1 << "." << endl;
    cout << "The integer with the greater value is: "
            << result2 << "." << endl;
    cout << endl;

    // Comparing the members of an initializer_list
    const int& result3 = max({ a, b });
    const int& result4 = max({ a, b }, abs_greater);
```

```
    cout << "Comparing the members of an initializer_list..." << endl;
    cout << "The member with the greater value is: " << result3 << endl;
    cout << "The integer with the greater absolute value is: " << result4 <<
endl;

    // Comparing set containers with elements of type CInt
    // using the max algorithm
    CInt c1 = 1, c2 = 2, c3 = 3;
    set<CInt> s1, s2, s3;
    set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s2.insert ( c2 );
    s2.insert ( c3 );

    cout << "s1 = (";
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ",";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    cout << "s2 = (";
    for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
        cout << " " << *s2_Iter << ",";
    s2_Iter = --s2.end( );
    cout << " " << *s2_Iter << " )." << endl;

    s3 = max ( s1, s2 );
    cout << "s3 = max ( s1, s2 ) = (";
    for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
        cout << " " << *s3_Iter << ",";
    s3_Iter = --s3.end( );
    cout << " " << *s3_Iter << " )." << endl << endl;

    // Comparing vectors with integer elements using the max algorithm
    vector<int> v1, v2, v3, v4, v5;
    vector<int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

    int i;
    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 0 ; ii <= 2 ; ii++ )
    {
        v2.push_back( ii );
    }

    int iii;
    for ( iii = 0 ; iii <= 2 ; iii++ )
    {
```

```
        v3.push_back( 2 * iii );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    cout << "Vector v3 is ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;

    v4 = max ( v1, v2 );
    v5 = max ( v1, v3 );

    cout << "Vector v4 = max (v1,v2) is ( " ;
    for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
        cout << *Iter4 << " ";
    cout << ")." << endl;

    cout << "Vector v5 = max (v1,v3) is ( " ;
    for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
        cout << *Iter5 << " ";
    cout << ")." << endl;
}
```

Output

```
Using integers 6 and -7...
The integer with the greater absolute value is: -7
The integer with the greater value is: 6.
Comparing the members of an initializer_list...
The member with the greater value is: 6
The integer with the greater absolute value is: -7
s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = max ( s1, s2 ) = ( CInt( 2 ), CInt( 3 ) ).

Vector v1 is ( 0 1 2 ).
Vector v2 is ( 0 1 2 ).
Vector v3 is ( 0 2 4 ).
Vector v4 = max (v1,v2) is ( 0 1 2 ).
Vector v5 = max (v1,v3) is ( 0 2 4 ).
```

max_element

Finds the first occurrence of largest element in a specified range where the ordering criterion may be specified by a binary predicate.

C++

```
template<class ForwardIterator>
constexpr ForwardIterator max_element(
    ForwardIterator first,
    ForwardIterator last );

template<class ForwardIterator, class Compare>
constexpr ForwardIterator max_element(
    ForwardIterator first,
    ForwardIterator last,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator max_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator max_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the range to be searched for the largest element.

last

A forward iterator addressing the position one past the final element in the range to be searched for the largest element.

pred

User-defined predicate function object that defines the sense in which one element is less than another. The comparison predicate takes two arguments and should return **true** when the first element is less than the second element and **false** otherwise.

Return value

A forward iterator addressing the position of the first occurrence of the largest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: `(last - first) - 1` comparisons are required for a nonempty range.

Example

C++

```
// alg_max_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<(ostream& osIn, const CInt& rhs)
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}
```

```
// Return whether modulus of elem1 is greater than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Searching a set container with elements of type CInt
    // for the maximum element
    CInt c1 = 1, c2 = 2, c3 = -3;
    set<CInt> s1;
    set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s1.insert ( c3 );

    cout << "s1 = (";
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ",";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    s1_R1_Iter = max_element ( s1.begin( ), s1.end( ) );

    cout << "The largest element in s1 is: " << *s1_R1_Iter << endl;
    cout << endl;

    // Searching a vector with elements of type int for the maximum
    // element under default less than & mod_lesser binary predicates
    vector<int> v1;
    vector<int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 1 ; ii <= 4 ; ii++ )
    {
        v1.push_back( - 2 * ii );
    }

    cout << "Vector v1 is ( " ;
    for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
        cout << *v1_Iter << " ";
    cout << ")." << endl;
```

```
    v1_R1_Iter = max_element ( v1.begin( ), v1.end( ) );
    v1_R2_Iter = max_element ( v1.begin( ), v1.end( ), mod_lesser);

    cout << "The largest element in v1 is: " << *v1_R1_Iter << endl;
    cout << "The largest element in v1 under the mod_lesser"
            << "\n binary predicate is: " << *v1_R2_Iter << endl;
}
```

Output

```
s1 = ( CInt( -3 ), CInt( 1 ), CInt( 2 ) ).
The largest element in s1 is: CInt( 2 )

Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 ).
The largest element in v1 is: 3
The largest element in v1 under the mod_lesser
 binary predicate is: -8
```

merge

Combines all of the elements from two sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

C++

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator merge(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator>
ForwardIterator merge(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
```

```
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator merge(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be combined and sorted into a single range.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be combined and sorted into a single range.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be combined and sorted into a single range.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be combined into a single sorted range.

pred

User-defined predicate function object that defines the sense in which one element is less than another. The comparison predicate takes two arguments and should return **`true`** when the first element is less than the second element, and **`false`** otherwise.

Return value

An output iterator addressing the position one past the last element in the sorted destination range.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range shouldn't overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges aren't modified by the algorithm `merge`.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range.

The complexity of the algorithm is linear with at most `(last1 - first1) - (last2 - first2) - 1` comparisons.

The list class provides a member function `merge` to merge the elements of two lists.

Example

C++

```
// alg_merge.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>   // For greater<int>( )
#include <iostream>
```

```
// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 ) {
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main() {
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1;

    // Constructing vector v1a and v1b with default less than ordering
    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1a.push_back( i );

    int ii;
    for ( ii =-5 ; ii <= 0 ; ii++ )
        v1b.push_back( ii );

    cout << "Original vector v1a with range sorted by the\n "
            << "binary predicate less than is v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
            << "binary predicate less than is v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vector v2 with ranges sorted by greater
    vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector<int>::iterator Iter2a, Iter2b, Iter2;
    sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
    sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );

    cout << "Original vector v2a with range sorted by the\n "
            << "binary predicate greater is   v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
            << "binary predicate greater is   v2b = ( " ;
    for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
        cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vector v3 with ranges sorted by mod_lesser
    vector<int> v3a( v1a ), v3b( v1b ) , v3( v1 );
```

```
    vector<int>::iterator Iter3a, Iter3b, Iter3;
    sort ( v3a.begin( ), v3a.end( ), mod_lesser );
    sort ( v3b.begin( ), v3b.end( ), mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
            << "binary predicate mod_lesser is   v3a = ( " ;
    for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
        cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
            << "binary predicate mod_lesser is   v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To merge inplace in ascending order with default binary
    // predicate less<int>( )
    merge ( v1a.begin( ), v1a.end( ), v1b.begin( ), v1b.end( ), v1.begin( )
);
    cout << "Merged inplace with default order,\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To merge inplace in descending order, specify binary
    // predicate greater<int>( )
    merge ( v2a.begin( ), v2a.end( ), v2b.begin( ), v2b.end( ),
        v2.begin( ), greater<int>( ) );
    cout << "Merged inplace with binary predicate greater specified,\n "
            << "vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // Applying A user-defined (UD) binary predicate mod_lesser
    merge ( v3a.begin( ), v3a.end( ), v3b.begin( ), v3b.end( ),
        v3.begin( ), mod_lesser );
    cout << "Merged inplace with binary predicate mod_lesser specified,\n "
            << "vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( 0 1 2 3 4 5 ).
Original vector v1b with range sorted by the
 binary predicate less than is v1b = ( -5 -4 -3 -2 -1 0 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a = ( 5 4 3 2 1 0 ).
```

```
Original vector v2b with range sorted by the
 binary predicate greater is   v2b = ( 0 -1 -2 -3 -4 -5 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a = ( 0 1 2 3 4 5 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b = ( 0 -1 -2 -3 -4 -5 ).
Merged inplace with default order,
 vector v1mod = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 ).
Merged inplace with binary predicate greater specified,
 vector v2mod = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 ).
Merged inplace with binary predicate mod_lesser specified,
 vector v3mod = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 ).
```

`min`

Compares two objects and returns the lesser of the two, where the ordering criterion may be specified by a binary predicate.

C++

```
template<class Type>
constexpr const Type& min(
    const Type& left,
    const Type& right);

template<class Type, class Pr>
constexpr const Type& min(
    const Type& left,
    const Type& right,
    BinaryPredicate pred);

template<class Type>
constexpr Type min(
    initializer_list<Type> ilist);

template<class Type, class Pr>
constexpr Type min(
    initializer_list<Type> ilist,
    BinaryPredicate pred);
```

Parameters

`left`

The first of the two objects being compared.

`right`

The second of the two objects being compared.

pred

A binary predicate used to compare the two objects.

inlist

The `initializer_list` that contains the members to be compared.

Return value

The lesser of the two objects, unless neither is lesser; in that case, it returns the first of the two objects. When an `initializer_list` is specified, it returns the least of the objects in the list.

Remarks

The `min` algorithm is unusual in having objects passed as parameters. Most C++ Standard Library algorithms operate on a range of elements whose position is specified by iterators passed as parameters. If you need a function that uses a range of elements, use min_element. constexpr was enabled on the `initializer_list` overloads in Visual Studio 2017.

Example

C++

```
// alg_min.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<(ostream& osIn, const CInt& rhs);
```

```
private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Comparing integers directly using the min algorithm with
    // binary predicate mod_lesser & with default less than
    int a = 6, b = -7, c = 7 ;
    const int& result1 = min ( a, b, mod_lesser );
    const int& result2 = min ( b, c );

    cout << "The mod_lesser of the integers 6 & -7 is: "
        << result1 << "." << endl;
    cout << "The lesser of the integers -7 & 7 is: "
        << result2 << "." << endl;
    cout << endl;

    // Comparing the members of an initializer_list
    const int& result3 = min({ a, c });
    const int& result4 = min({ a, b }, mod_lesser);

    cout << "The lesser of the integers 6 & 7 is: "
        << result3 << "." << endl;
    cout << "The mod_lesser of the integers 6 & -7 is: "
        << result4 << "." << endl;
    cout << endl;

    // Comparing set containers with elements of type CInt
    // using the min algorithm
    CInt c1 = 1, c2 = 2, c3 = 3;
    set<CInt> s1, s2, s3;
    set<CInt>::iterator s1_Iter, s2_Iter, s3_Iter;

    s1.insert ( c1 );
    s1.insert ( c2 );
    s2.insert ( c2 );
    s2.insert ( c3 );
```

```
cout << "s1 = (";
for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
    cout << " " << *s1_Iter << ",";
s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

cout << "s2 = (";
for ( s2_Iter = s2.begin( ); s2_Iter != --s2.end( ); s2_Iter++ )
    cout << " " << *s2_Iter << ",";
s2_Iter = --s2.end( );
cout << " " << *s2_Iter << " )." << endl;

s3 = min ( s1, s2 );
cout << "s3 = min ( s1, s2 ) = (";
for ( s3_Iter = s3.begin( ); s3_Iter != --s3.end( ); s3_Iter++ )
    cout << " " << *s3_Iter << ",";
s3_Iter = --s3.end( );
cout << " " << *s3_Iter << " )." << endl << endl;

// Comparing vectors with integer elements using min algorithm
vector<int> v1, v2, v3, v4, v5;
vector<int>::iterator Iter1, Iter2, Iter3, Iter4, Iter5;

int i;
for ( i = 0 ; i <= 2 ; i++ )
{
    v1.push_back( i );
}

int ii;
for ( ii = 0 ; ii <= 2 ; ii++ )
{
    v2.push_back( ii );
}

int iii;
for ( iii = 0 ; iii <= 2 ; iii++ )
{
    v3.push_back( 2 * iii );
}

cout << "Vector v1 is ( " ;
for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
    cout << *Iter1 << " ";
cout << ")." << endl;

cout << "Vector v2 is ( " ;
for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
    cout << *Iter2 << " ";
cout << ")." << endl;

cout << "Vector v3 is ( " ;
for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
    cout << *Iter3 << " ";
```

```
    cout << ")." << endl;

    v4 = min ( v1, v2 );
    v5 = min ( v1, v3 );

    cout << "Vector v4 = min ( v1,v2 ) is ( " ;
    for ( Iter4 = v4.begin( ) ; Iter4 != v4.end( ) ; Iter4++ )
        cout << *Iter4 << " ";
    cout << ")." << endl;

    cout << "Vector v5 = min ( v1,v3 ) is ( " ;
    for ( Iter5 = v5.begin( ) ; Iter5 != v5.end( ) ; Iter5++ )
        cout << *Iter5 << " ";
    cout << ")." << endl;
}
```

Output

```
The mod_lesser of the integers 6 & -7 is: 6.
The lesser of the integers -7 & 7 is: -7.
The lesser of the integers 6 & 7 is: 6.The mod_lesser of the integers 6 & -7
is: 6.
s1 = ( CInt( 1 ), CInt( 2 ) ).
s2 = ( CInt( 2 ), CInt( 3 ) ).
s3 = min ( s1, s2 ) = ( CInt( 1 ), CInt( 2 ) ).

Vector v1 is ( 0 1 2 ).
Vector v2 is ( 0 1 2 ).
Vector v3 is ( 0 2 4 ).
Vector v4 = min ( v1,v2 ) is ( 0 1 2 ).
Vector v5 = min ( v1,v3 ) is ( 0 1 2 ).
```

`min_element`

Finds the first occurrence of smallest element in a specified range where the ordering criterion may be specified by a binary predicate.

C++

```
template<class ForwardIterator>
constexpr ForwardIterator min_element(
    ForwardIterator first,
    ForwardIterator last );

template<class ForwardIterator, class Compare>
constexpr ForwardIterator min_element(
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

```
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator min_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator min_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the range to be searched for the smallest element.

last

A forward iterator addressing the position one past the final element in the range to be searched for the smallest element.

pred

User-defined predicate function object that defines the sense in which one element is less than another. The comparison predicate takes two arguments and should return **`true`** when the first element is less than the second element, and **`false`** otherwise.

Return value

A forward iterator addressing the position of the first occurrence of the smallest element in the range searched.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

The complexity is linear: `(last - first) - 1` comparisons are required for a nonempty range.

Example

C++

```
// alg_min_element.cpp
// compile with: /EHsc
#include <vector>
#include <set>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt& operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Searching a set container with elements of type CInt
    // for the minimum element
    CInt c1 = 1, c2 = 2, c3 = -3;
    set<CInt> s1;
    set<CInt>::iterator s1_Iter, s1_R1_Iter, s1_R2_Iter;
```

```
    s1.insert ( c1 );
    s1.insert ( c2 );
    s1.insert ( c3 );

    cout << "s1 = (";
    for ( s1_Iter = s1.begin( ); s1_Iter != --s1.end( ); s1_Iter++ )
        cout << " " << *s1_Iter << ",";
    s1_Iter = --s1.end( );
    cout << " " << *s1_Iter << " )." << endl;

    s1_R1_Iter = min_element ( s1.begin( ), s1.end( ) );

    cout << "The smallest element in s1 is: " << *s1_R1_Iter << endl;
    cout << endl;

    // Searching a vector with elements of type int for the maximum
    // element under default less than & mod_lesser binary predicates
    vector<int> v1;
    vector<int>::iterator v1_Iter, v1_R1_Iter, v1_R2_Iter;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii = 1 ; ii <= 4 ; ii++ )
    {
        v1.push_back( - 2 * ii );
    }

    cout << "Vector v1 is ( " ;
    for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
        cout << *v1_Iter << " ";
    cout << ")." << endl;

    v1_R1_Iter = min_element ( v1.begin( ), v1.end( ) );
    v1_R2_Iter = min_element ( v1.begin( ), v1.end( ), mod_lesser);

    cout << "The smallest element in v1 is: " << *v1_R1_Iter << endl;
    cout << "The smallest element in v1 under the mod_lesser"
        << "\n binary predicate is: " << *v1_R2_Iter << endl;
}
```

Output

```
s1 = ( CInt( -3 ), CInt( 1 ), CInt( 2 ) ).
The smallest element in s1 is: CInt( -3 )

Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 ).
The smallest element in v1 is: -8
```

```
The smallest element in v1 under the mod_lesser
binary predicate is: 0
```

`minmax_element`

Performs the work performed by `min_element` and `max_element` in one call.

C++

```
template<class ForwardIterator>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class Compare>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);

template<class ExecutionPolicy, class ForwardIterator>
pair<ForwardIterator, ForwardIterator> minmax_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator that indicates the beginning of a range.

last

A forward iterator that indicates the end of a range.

pred

A user-defined predicate function object that defines the sense in which one element is

less than another. The comparison predicate takes two arguments and should return `true` when the first is less than the second, and `false` otherwise.

Return value

Returns

`pair<ForwardIterator, ForwardIterator>( min_element(first, last), max_element(first, last))`.

Remarks

The first template function returns

`pair<ForwardIterator,ForwardIterator>(min_element(first,last), max_element(first,last))`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pred(X, Y)`.

If the sequence is non-empty, the function performs at most `3 * (last - first - 1) / 2` comparisons.

`minmax`

Compares two input parameters and returns them as a pair, in order of lesser to greater.

C++

```
template<class Type>
constexpr pair<const Type&, const Type&> minmax(
    const Type& left,
    const Type& right);

template<class Type, class BinaryPredicate>
constexpr pair<const Type&, const Type&> minmax(
    const Type& left,
    const Type& right,
    BinaryPredicate pred);

template<class Type>
constexpr pair<Type&, Type&> minmax(
    initializer_list<Type> ilist);

template<class Type, class BinaryPredicate>
constexpr pair<Type&, Type&> minmax(
```

```
    initializer_list<Type> ilist,
    BinaryPredicate pred);
```

Parameters

left

The first of the two objects being compared.

right

The second of the two objects being compared.

pred

A binary predicate used to compare the two objects.

inlist

The `initializer_list` that contains the members to be compared.

Remarks

The first template function returns `pair<const Type&, const Type&>( right, left )` if *right* is less than *left*. Otherwise, it returns `pair<const Type&, const Type&>( left, right )`.

The second member function returns a pair where the first element is the lesser and the second is the greater when compared by the predicate *pred*.

The remaining template functions behave the same, except that they replace the *left* and *right* parameters with *inlist*.

The function performs exactly one comparison.

`mismatch`

Compares two ranges element by element and locates the first position where a difference occurs.

Use the dual-range overloads in C++14 code because the overloads that only take a single iterator for the second range won't detect differences if the second range is longer than the first range. These overloads will result in undefined behavior if the second range is shorter than the first range.

```
C++
```

```
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2 );

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    BinaryPredicate pred );

//C++14
template<class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2 );

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2>>
mismatch(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    BinaryPredicate pred);

//C++17
template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
pair<ForwardIterator1, ForwardIterator2>
mismatch(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class BinaryPredicate>
pair<ForwardIterator1, ForwardIterator2>
mismatch(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
```

```
    ForwardIterator2>
    pair<ForwardIterator1, ForwardIterator2>
    mismatch(
        ExecutionPolicy&& exec,
        ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2,
        ForwardIterator2 last2);

    template<class ExecutionPolicy, class ForwardIterator1, class
    ForwardIterator2, class BinaryPredicate>
    pair<ForwardIterator1, ForwardIterator2>
    mismatch(
        ExecutionPolicy&& exec,
        ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2,
        ForwardIterator2 last2,
        BinaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first range to be tested.

last1

An input iterator addressing the position one past the last element in the first range to be tested.

first2

An input iterator addressing the position of the first element in the second range to be tested.

last2

An input iterator addressing the position of one past the last element in the second range to be tested.

pred

User-defined predicate function object that compares the current elements in each range and determines whether they're equivalent. It returns **true** when satisfied and **false** when not satisfied.

Return value

Returns a pair of iterators addressing the positions of the mismatch in the two ranges. The first component iterator points to the position in the first range. The second component iterator points to the position in the second range. If there's no difference between the elements in the ranges compared or if the binary predicate in the second version is satisfied by all element pairs from the two ranges, then the first component iterator points to the position one past the final element in the first range and the second component iterator points to a position one past the final element tested in the second range.

Remarks

The first template function assumes that there are as many elements in the range beginning at first2 as there are in the range designated by [first1, last1). If there are more in the second range, they're ignored; if there are less, then undefined behavior will result.

The range to be searched must be valid; all iterators must be dereferenceable and the last position is reachable from the first by incrementation.

The time complexity of the algorithm is linear in the number of elements contained in the shorter range.

The user-defined predicate isn't required to impose an equivalence relation that symmetric, reflexive, and transitive between its operands.

Example

The following example demonstrates how to use mismatch. The C++03 overload is shown only in order to demonstrate how it can produce an unexpected result.

C++

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>

using namespace std;

// Return whether first element is twice the second
// Note that this isn't a symmetric, reflexive, and transitive equivalence.
```

```
// mismatch and equal accept such predicates, but is_permutation doesn't.
bool twice(int elem1, int elem2)
{
    return elem1 == elem2 * 2;
}

void PrintResult(const string& msg, const pair<vector<int>::iterator,
vector<int>::iterator>& result,
    const vector<int>& left, const vector<int>& right)
{
    // If either iterator stops before reaching the end of its container,
    // it means a mismatch was detected.
    if (result.first != left.end() || result.second != right.end())
    {
        string leftpos(result.first == left.end() ? "end" :
to_string(*result.first));
        string rightpos(result.second == right.end() ? "end" :
to_string(*result.second));
        cout << msg << "mismatch. Left iterator at " << leftpos
            << " right iterator at " << rightpos << endl;
    }
    else
    {
        cout << msg << " match." << endl;
    }
}

int main()
{

    vector<int> vec_1{ 0, 5, 10, 15, 20, 25 };
    vector<int> vec_2{ 0, 5, 10, 15, 20, 25, 30 };

    // Testing different length vectors for mismatch (C++03)
    auto match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin());
    bool is_mismatch = match_vecs.first != vec_1.end();
    cout << "C++03: vec_1 and vec_2 are a mismatch: " << boolalpha <<
is_mismatch << endl;

    // Using dual-range overloads:

    // Testing different length vectors for mismatch (C++14)
    match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin(),
vec_2.end());
    PrintResult("C++14: vec_1 and vec_2: ", match_vecs, vec_1, vec_2);

    // Identify point of mismatch between vec_1 and modified vec_2.
    vec_2[3] = 42;
    match_vecs = mismatch(vec_1.begin(), vec_1.end(), vec_2.begin(),
vec_2.end());
    PrintResult("C++14 vec_1 v. vec_2 modified: ", match_vecs, vec_1,
vec_2);

    // Test vec_3 and vec_4 for mismatch under the binary predicate twice
(C++14)
```

```
    vector<int> vec_3{ 10, 20, 30, 40, 50, 60 };
    vector<int> vec_4{ 5, 10, 15, 20, 25, 30 };
    match_vecs = mismatch(vec_3.begin(), vec_3.end(), vec_4.begin(),
vec_4.end(), twice);
    PrintResult("vec_3 v. vec_4 with pred: ", match_vecs, vec_3, vec_4);

    vec_4[5] = 31;
    match_vecs = mismatch(vec_3.begin(), vec_3.end(), vec_4.begin(),
vec_4.end(), twice);
    PrintResult("vec_3 v. modified vec_4 with pred: ", match_vecs, vec_3,
vec_4);

    // Compare a vector<int> to a list<int>
    list<int> list_1{ 0, 5, 10, 15, 20, 25 };
    auto match_vec_list = mismatch(vec_1.begin(), vec_1.end(),
list_1.begin(), list_1.end());
    is_mismatch = match_vec_list.first != vec_1.end() ||
match_vec_list.second != list_1.end();
    cout << "vec_1 and list_1 are a mismatch: " << boolalpha << is_mismatch
<< endl;

    char c;
    cout << "Press a key" << endl;
    cin >> c;

}
```

Output

```
C++03: vec_1 and vec_2 are a mismatch: false
C++14: vec_1 and vec_2: mismatch. Left iterator at end right iterator at 30
C++14 vec_1 v. vec_2 modified: mismatch. Left iterator at 15 right iterator
at 42
C++14 vec_3 v. vec_4 with pred: match.
C++14 vec_3 v. modified vec_4 with pred: mismatch. Left iterator at 60 right
iterator at 31
C++14: vec_1 and list_1 are a mismatch: false
Press a key
```

<alg> move

Move elements associated with a specified range.

C++

```
template<class InputIterator, class OutputIterator>
OutputIterator move(
    InputIterator first,
    InputIterator last,
    OutputIterator dest);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 move(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result);
```

Parameters

exec

The execution policy to use.

first

An input iterator that indicates where to start the range of elements to move.

last

An input iterator that indicates the end of a range of elements to move.

dest

The output iterator that is to contain the moved elements.

Remarks

The template function evaluates `*(dest + N) = move(*(first + N))` once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. It then returns `dest + N`. If `dest` and *first* designate regions of storage, *dest* must not be in the range `[first, last)`.

move_backward

Moves the elements of one iterator to another. The move starts with the last element in a specified range, and ends with the first element in that range.

C++

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 move_backward(
    BidirectionalIterator1 first,
    BidirectionalIterator1 last,
    BidirectionalIterator2 destEnd);
```

Parameters

first

An iterator that indicates the start of a range to move elements from.

last

An iterator that indicates the end of a range to move elements from. This element isn't moved.

destEnd

A bidirectional iterator addressing the position of one past the final element in the destination range.

Remarks

The template function evaluates `*(destEnd - N - 1) = move(*(last - N - 1))` once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` starting with the lowest value. It then returns `destEnd - (last - first)`. If *destEnd* and *first* designate regions of storage, *destEnd* must not be in the range `[first, last)`.

`move` and `move_backward` are functionally equivalent to using `copy` and `copy_backward` with a move iterator.

`next_permutation`

Reorders the elements in a range so that the original ordering is replaced by the lexicographically next greater permutation if it exists. The sense of lexicographically next may be specified with a binary predicate.

C++

```
template<class BidirectionalIterator>
bool next_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last);

template<class BidirectionalIterator, class BinaryPredicate>
bool next_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last,
    BinaryPredicate pred);
```

Parameters

first

A bidirectional iterator pointing to the position of the first element in the range to be permuted.

last

A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

pred

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

true if the lexicographically next permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically smallest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less than comparable to ensure that the next permutation is well defined.

The complexity is linear with at most `(last - first) / 2` swaps.

Example

C++

```
// alg_next_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );
```

```
class CInt
{
public:
    CInt( int n = 0 ) : m_nVal( n ) {}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ) {}
    CInt& operator=( const CInt& rhs ) {m_nVal =
        rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        { return ( m_nVal < rhs.m_nVal ); }
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs )
{
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Reordering the elements of type CInt in a deque
    // using the prev_permutation algorithm
    CInt c1 = 5, c2 = 1, c3 = 10;
    bool deq1Result;
    deque<CInt> deq1, deq2, deq3;
    deque<CInt>::iterator d1_Iter;

    deq1.push_back ( c1 );
    deq1.push_back ( c2 );
    deq1.push_back ( c3 );

    cout << "The original deque of CInts is deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    deq1Result = next_permutation ( deq1.begin( ), deq1.end( ) );

    if ( deq1Result )
        cout << "The lexicographically next permutation "
```

```
            << "exists and has\nreplaced the original "
            << "ordering of the sequence in deq1." << endl;
    else
        cout << "The lexicographically next permutation doesn't "
            << "exist\n and the lexicographically "
            << "smallest permutation\n has replaced the "
            << "original ordering of the sequence in deq1." << endl;

    cout << "After one application of next_permutation,\n deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl << endl;

    // Permuting vector elements with binary function mod_lesser
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = -3 ; i <= 3 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    next_permutation ( v1.begin( ), v1.end( ), mod_lesser );

    cout << "After the first next_permutation, vector v1 is:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= 5 ) {
        next_permutation ( v1.begin( ), v1.end( ), mod_lesser );
        cout << "After another next_permutation of vector v1,\n v1 =   ( " ;
        for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ;Iter1 ++ )
            cout << *Iter1 << " ";
        cout << ")." << endl;
        iii++;
    }
}
```

Output

```
The original deque of CInts is deq1 = ( CInt( 5 ), CInt( 1 ), CInt( 10 ) ).
The lexicographically next permutation exists and has
replaced the original ordering of the sequence in deq1.
After one application of next_permutation,
```

```
deq1 = ( CInt( 5 ), CInt( 10 ), CInt( 1 ) ).

Vector v1 is ( -3 -2 -1 0 1 2 3 ).
After the first next_permutation, vector v1 is:
v1 = ( -3 -2 -1 0 1 3 2 ).
After another next_permutation of vector v1,
v1 =   ( -3 -2 -1 0 2 1 3 ).
After another next_permutation of vector v1,
v1 =   ( -3 -2 -1 0 2 3 1 ).
After another next_permutation of vector v1,
v1 =   ( -3 -2 -1 0 3 1 2 ).
After another next_permutation of vector v1,
v1 =   ( -3 -2 -1 0 3 2 1 ).
After another next_permutation of vector v1,
v1 =   ( -3 -2 -1 1 0 2 3 ).
```

nth_element

Partitions a range of elements, correctly locating the *n*th element of the sequence in the range that meets these criteria: All the elements in front of it are less than or equal to it, and all the elements that follow it are greater than or equal to it.

C++

```
template<class RandomAccessIterator>
void nth_element(
    RandomAccessIterator first,
    RandomAccessIterator nth,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(
    RandomAccessIterator first,
    RandomAccessIterator nth,
    RandomAccessIterator last,
    Compare pred);

template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator nth,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void nth_element(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator nth,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

exec
The execution policy to use.

first
A random-access iterator addressing the position of the first element in the range to be partitioned.

nth
A random-access iterator addressing the position of element to be correctly ordered on the boundary of the partition.

last
A random-access iterator addressing the position one past the final element in the range to be partitioned.

pred
User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A comparison predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The `nth_element` algorithm doesn't guarantee that elements in the subranges either side of the *n*th element are sorted. It thus makes fewer guarantees than `partial_sort`, which orders the elements in the range below some chosen element, and may be used as a faster alternative to `partial_sort` when the ordering of the lower range isn't required.

Elements are equivalent, but not necessarily equal, if neither is less than the other.

The average of a sort complexity is linear with respect to *last - first*.

Example

```
C++
```

```
// alg_nth_elem.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 ) {
    return elem1 > elem2;
}

int main() {
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
        v1.push_back( 3 * i );

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
        v1.push_back( 3 * ii + 1 );

    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
        v1.push_back( 3 * iii +2 );

    cout << "Original vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    nth_element(v1.begin( ), v1.begin( ) + 3, v1.end( ) );
    cout << "Position 3 partitioned vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order, specify binary predicate
    nth_element( v1.begin( ), v1.begin( ) + 4, v1.end( ),
            greater<int>( ) );
    cout << "Position 4 partitioned (greater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    random_shuffle( v1.begin( ), v1.end( ) );
    cout << "Shuffled vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
```

```
    // A user-defined (UD) binary predicate can also be used
    nth_element( v1.begin( ), v1.begin( ) + 5, v1.end( ), UDgreater );
    cout << "Position 5 partitioned (UDgreater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

Output

```
 v1 = ( 0 3 6 9 12 15 1 4 7 10 13 16 2 5 8 11 14 17 )
Position 3 partitioned vector:
 v1 = ( 1 0 2 3 8 5 9 4 7 6 10 16 13 15 12 11 14 17 )
Position 4 partitioned (greater) vector:
 v1 = ( 16 17 14 15 13 12 11 9 7 8 10 6 1 4 5 3 2 0 )
Shuffled vector:
 v1 = ( 13 10 6 3 5 2 0 17 11 8 15 9 7 14 16 1 12 4 )
Position 5 partitioned (UDgreater) vector:
 v1 = ( 14 17 15 16 13 12 10 11 9 8 0 2 7 5 3 1 6 4 )
```

none_of

Returns **`true`** when a condition is never present among elements in the given range.

C++

```
template<class InputIterator, class UnaryPredicate>
bool none_of(
    InputIterator first,
    InputIterator last,
    UnaryPredicate pred);

template <class ExecutionPolicy, class ForwardIterator, class
UnaryPredicate>
bool none_of(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

`exec`

The execution policy to use.

first

An input iterator that indicates where to start to check a range of elements for a condition.

last

An input iterator that indicates the end of a range of elements.

pred

The condition to test for. This test is provided by a user-defined predicate function object that defines the condition. A unary predicate takes a single argument and returns **true** or **false**.

Return value

Returns **true** if the condition isn't detected at least once in the indicated range, and **false** if the condition is detected.

Remarks

The template function returns **true** only if, for some `N` in the range `[0, last - first)`, the predicate `pred(*(first + N))` is always **false**.

partial_sort

Arranges a specified number of the smaller elements in a range into a nondescending order. A binary predicate may supply an ordering criterion.

C++

```
template<class RandomAccessIterator>
void partial_sort(
    RandomAccessIterator first,
    RandomAccessIterator sortEnd,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(
    RandomAccessIterator first,
    RandomAccessIterator sortEnd,
    RandomAccessIterator last
    Compare pred);

template<class ExecutionPolicy, class RandomAccessIterator>
void partial_sort(
    ExecutionPolicy&& exec,
```

```
    RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void partial_sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A random-access iterator addressing the position of the first element in the range to be sorted.

sortEnd

A random-access iterator addressing the position one past the final element in the subrange to be sorted.

last

A random-access iterator addressing the position one past the final element in the range to be partially sorted.

pred

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm isn't stable and doesn't guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_sort` does preserve this original ordering.

The average partial sort complexity is *O*((`last` - `first`) log (`sortEnd` - `first`)).

Example

C++

```
// alg_partial_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
    {
        v1.push_back( 2 * ii +1 );
    }

    cout << "Original vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    partial_sort(v1.begin( ), v1.begin( ) + 6, v1.end( ) );
    cout << "Partially sorted vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To partially sort in descending order, specify binary predicate
    partial_sort(v1.begin( ), v1.begin( ) + 4, v1.end( ), greater<int>( ) );
    cout << "Partially resorted (greater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
```

```
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    partial_sort(v1.begin( ), v1.begin( ) + 8, v1.end( ), UDgreater );
    cout << "Partially resorted (UDgreater) vector:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

Output

```
Original vector:
v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Partially sorted vector:
v1 = ( 0 1 2 3 4 5 10 8 6 7 9 11 )
Partially resorted (greater) vector:
v1 = ( 11 10 9 8 0 1 2 3 4 5 6 7 )
Partially resorted (UDgreater) vector:
v1 = ( 11 10 9 8 7 6 5 4 0 1 2 3 )
```

partial_sort_copy

Copies elements from a source range into a destination range where the source elements are ordered by either less than or another specified binary predicate.

C++

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    InputIterator first1,
    InputIterator last1,
    RandomAccessIterator first2,
    RandomAccessIterator last2 );

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
    InputIterator first1,
    InputIterator last1,
    RandomAccessIterator first2,
    RandomAccessIterator last2,
    Compare pred);

template<class ExecutionPolicy, class ForwardIterator, class
RandomAccessIterator>
RandomAccessIterator partial_sort_copy(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    RandomAccessIterator result_first,
```

```
    RandomAccessIterator result_last);

template<class ExecutionPolicy, class ForwardIterator, class
RandomAccessIterator, class Compare>
RandomAccessIterator partial_sort_copy(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    RandomAccessIterator result_first,
    RandomAccessIterator result_last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the source range.

last1

An input iterator addressing the position one past the final element in the source range.

first2

A random-access iterator addressing the position of the first element in the sorted destination range.

last2

A random-access iterator addressing the position one past the final element in the sorted destination range.

pred

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

A random-access iterator addressing the element in the destination range one position beyond the last element inserted from the source range.

Remarks

The source and destination ranges must not overlap and must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The binary predicate must provide a strict weak ordering so that elements that aren't equivalent are ordered, but elements that are equivalent aren't. Two elements are equivalent under less than, but not necessarily equal, if neither is less than the other.

Example

C++

```
// alg_partial_sort_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1, v2;
    list<int> list1;
    vector<int>::iterator iter1, iter2;
    list<int>::iterator list1_Iter, list1_inIter;

    int i;
    for (i = 0; i <= 9; i++)
        v1.push_back(i);

    random_shuffle(v1.begin(), v1.end());

    list1.push_back(60);
    list1.push_back(50);
    list1.push_back(20);
    list1.push_back(30);
    list1.push_back(40);
    list1.push_back(10);

    cout << "Vector v1 = ( " ;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;

    cout << "List list1 = ( " ;
    for (list1_Iter = list1.begin();
         list1_Iter!= list1.end();
         list1_Iter++)
        cout << *list1_Iter << " ";
    cout << ")" << endl;
```

```
    // Copying a partially sorted copy of list1 into v1
    vector<int>::iterator result1;
    result1 = partial_sort_copy(list1.begin(), list1.end(),
             v1.begin(), v1.begin() + 3);

    cout << "List list1 Vector v1 = ( " ;
    for (iter1 = v1.begin() ; iter1 != v1.end() ; iter1++)
        cout << *iter1 << " ";
    cout << ")" << endl;
    cout << "The first v1 element one position beyond"
         << "\n the last L 1 element inserted was " << *result1
         << "." << endl;

    // Copying a partially sorted copy of list1 into v2
    int ii;
    for (ii = 0; ii <= 9; ii++)
        v2.push_back(ii);

    random_shuffle(v2.begin(), v2.end());
    vector<int>::iterator result2;
    result2 = partial_sort_copy(list1.begin(), list1.end(),
             v2.begin(), v2.begin() + 6);

    cout << "List list1 into Vector v2 = ( " ;
    for (iter2 = v2.begin() ; iter2 != v2.end(); iter2++)
        cout << *iter2 << " ";
    cout << ")" << endl;
    cout << "The first v2 element one position beyond"
         << "\n the last L 1 element inserted was " << *result2
         << "." << endl;
}
```

Output

```
Vector v1 = ( 4 3 7 8 0 5 2 1 6 9 )
List list1 = ( 60 50 20 30 40 10 )
List list1 Vector v1 = ( 10 20 30 8 0 5 2 1 6 9 )
The first v1 element one position beyond
 the last L 1 element inserted was 8.
List list1 into Vector v2 = ( 10 20 30 40 50 60 9 6 7 8 )
The first v2 element one position beyond
 the last L 1 element inserted was 9.
```

`partition`

Classifies elements in a range into two disjoint sets, with those elements satisfying a unary predicate preceding the ones that fail to satisfy it.

C++

```
template<class BidirectionalIterator, class UnaryPredicate>
BidirectionalIterator partition(
    BidirectionalIterator first,
    BidirectionalIterator last,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate>
ForwardIterator partition(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A unary predicate takes a single argument and returns **true** or **false**.

Return value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements *a* and *b* are equivalent, but not necessarily equal, if both `pred( a, b )` is false and `pred( b, a )` is false, where *pred* is the parameter-specified predicate. The

`partition` algorithm isn't stable and doesn't guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_partition` does preserve this original ordering.

The complexity is linear: there are `(last - first)` applications of *`pred`* and at most `(last - first)/2` swaps.

Example

C++

```
// alg_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value )
{
    return value > 5;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 10 ; i++ )
    {
        v1.push_back( i );
    }
    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Partition the range with predicate greater10
    partition ( v1.begin( ), v1.end( ), greater5 );
    cout << "The partitioned set of elements in v1 is: ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 4 10 7 8 0 5 2 1 6 9 3 ).
The partitioned set of elements in v1 is: ( 9 10 7 8 6 5 2 1 0 4 3 ).
```

`partition_copy`

Copies elements for which a condition is **`true`** to one destination, and for which the condition is **`false`** to another. The elements must come from a specified range.

C++

```
template<class InputIterator, class OutputIterator1, class OutputIterator2,
class UnaryPredicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator1 dest1,
    OutputIterator2 dest2,
    UnaryPredicate pred);

template <class ExecutionPolicy, class ForwardIterator, class
ForwardIterator1, class ForwardIterator2, class UnaryPredicate>
pair<ForwardIterator1, ForwardIterator2> partition_copy(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    ForwardIterator1 out_true,
    ForwardIterator2 out_false,
    UnaryPredicate pred);
```

Parameters

`exec`

The execution policy to use.

`first`

An input iterator that indicates the beginning of a range to check for a condition.

`last`

An input iterator that indicates the end of a range.

`dest1`

An output iterator used to copy elements that return true for a condition tested by using *`pred`*.

dest2

An output iterator used to copy elements that return false for a condition tested by using *pred*.

pred

The condition to test for. The test is provided by a user-defined predicate function object that defines the condition to be tested. A unary predicate takes a single argument and returns **true** or **false**.

Remarks

The template function copies each element `X` in `[first,last)` to `*dest1++` if `pred(X)` is true, or to `*dest2++` if not. It returns `pair<OutputIterator1, OutputIterator2>(dest1, dest2)`.

partition_point

Returns the first element in the given range that doesn't satisfy the condition. The elements are sorted so that the ones that satisfy the condition come before the ones that don't.

```
C++
```

```
template<class ForwardIterator, class UnaryPredicate>
ForwardIterator partition_point(
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

first

A `ForwardIterator` that indicates the start of a range to check for a condition.

last

A `ForwardIterator` that indicates the end of a range.

pred

The condition to test for. The test is provided by a user-defined predicate function object that defines the condition to be satisfied by the element being searched for. A unary predicate takes a single argument and returns **true** or **false**.

Return value

Returns a `ForwardIterator` that refers to the first element that doesn't fulfill the condition tested for by *`pred`*, or returns *`last`* if one isn't found.

Remarks

The template function finds the first iterator `it` in `[first, last)` for which `pred(*it)` is **`false`**. The sequence must be ordered by *`pred`*.

pop_heap

Removes the largest element from the front of a heap to the next-to-last position in the range and then forms a new heap from the remaining elements.

C++

```
template<class RandomAccessIterator>
void pop_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class BinaryPredicate>
void pop_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    BinaryPredicate pred);
```

Parameters

`first`
A random-access iterator addressing the position of the first element in the heap.

`last`
A random-access iterator addressing the position one past the final element in the heap.

`pred`
User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **`true`** when satisfied and **`false`** when not satisfied.

Remarks

The `pop_heap` algorithm is the inverse of the operation performed by the push_heap algorithm, in which an element at the next-to-last position of a range is added to a heap consisting of the prior elements in the range, in the case when the element being added to the heap is larger than any of the elements already in the heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they're used in the implementation of the C++ Standard Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most `log (last - first)` comparisons.

Example

C++

```
// alg_pop_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 1 ; i <= 9 ; i++ )
        v1.push_back( i );

    // Make v1 a heap with default less than ordering
    random_shuffle( v1.begin( ), v1.end( ) );
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
```

```
    // Add an element to the back of the heap
    v1.push_back( 10 );
    push_heap( v1.begin( ), v1.end( ) );
    cout << "The reheaped v1 with 10 added is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove the largest element from the heap
    pop_heap( v1.begin( ), v1.end( ) );
    cout << "The heap v1 with 10 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl << endl;

    // Make v1 a heap with greater-than ordering with a 0 element
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    v1.push_back( 0 );
    push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The 'greater than' reheaped v1 puts the smallest "
        << "element first:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Application of pop_heap to remove the smallest element
    pop_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The 'greater than' heaped v1 with the smallest element\n "
        << "removed from the heap is: ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
The heaped version of vector v1 is ( 9 7 8 5 1 6 3 2 4 ).
The reheaped v1 with 10 added is ( 10 9 8 5 7 6 3 2 4 1 ).
The heap v1 with 10 removed is ( 9 7 8 5 1 6 3 2 4 10 ).

The 'greater than' reheaped v1 puts the smallest element first:
 ( 0 1 3 4 2 6 8 5 9 10 7 ).
The 'greater than' heaped v1 with the smallest element
 removed from the heap is: ( 1 2 3 4 7 6 8 5 9 10 0 ).
```

prev_permutation

Reorders the elements in a range so that the original ordering is replaced by the lexicographically previous greater permutation if it exists. A binary predicate may specify

the sense of lexicographically previous.

```
C++
```

```
template<class BidirectionalIterator>
bool prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last);

template<class BidirectionalIterator, class BinaryPredicate>
bool prev_permutation(
    BidirectionalIterator first,
    BidirectionalIterator last,
    BinaryPredicate pred);
```

Parameters

first
A bidirectional iterator pointing to the position of the first element in the range to be permuted.

last
A bidirectional iterator pointing to the position one past the final element in the range to be permuted.

pred
User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

true if the lexicographically previous permutation exists and has replaced the original ordering of the range; otherwise **false**, in which case the ordering is transformed into the lexicographically largest permutation.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The default binary predicate is less than and the elements in the range must be less-than comparable to ensure that the previous permutation is well defined.

The complexity is linear, with at most (`last` - `first`)/2 swaps.

Example

C++

```
// alg_prev_perm.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <ostream>

using namespace std;
class CInt;
ostream& operator<<( ostream& osIn, const CInt& rhs );

class CInt {
public:
    CInt( int n = 0 ) : m_nVal( n ){}
    CInt( const CInt& rhs ) : m_nVal( rhs.m_nVal ){}
    CInt&   operator=( const CInt& rhs ) {m_nVal =
    rhs.m_nVal; return *this;}
    bool operator<( const CInt& rhs ) const
        {return ( m_nVal < rhs.m_nVal );}
    friend ostream& operator<<( ostream& osIn, const CInt& rhs );

private:
    int m_nVal;
};

inline ostream& operator<<( ostream& osIn, const CInt& rhs ) {
    osIn << "CInt( " << rhs.m_nVal << " )";
    return osIn;
}

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 ) {
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
};

int main()
{
    // Reordering the elements of type CInt in a deque
    // using the prev_permutation algorithm
    CInt c1 = 1, c2 = 5, c3 = 10;
    bool deq1Result;
    deque<CInt> deq1, deq2, deq3;
```

```
    deque<CInt>::iterator d1_Iter;

    deq1.push_back ( c1 );
    deq1.push_back ( c2 );
    deq1.push_back ( c3 );

    cout << "The original deque of CInts is deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl;

    deq1Result = prev_permutation ( deq1.begin( ), deq1.end( ) );

    if ( deq1Result )
        cout << "The lexicographically previous permutation "
            << "exists and has \nreplaced the original "
            << "ordering of the sequence in deq1." << endl;
    else
        cout << "The lexicographically previous permutation doesn't "
            << "exist\n and the lexicographically "
            << "smallest permutation\n has replaced the "
            << "original ordering of the sequence in deq1." << endl;

    cout << "After one application of prev_permutation,\n deq1 = (";
    for ( d1_Iter = deq1.begin( ); d1_Iter != --deq1.end( ); d1_Iter++ )
        cout << " " << *d1_Iter << ",";
    d1_Iter = --deq1.end( );
    cout << " " << *d1_Iter << " )." << endl << endl;

    // Permutating vector elements with binary function mod_lesser
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = -3 ; i <= 3 ; i++ )
        v1.push_back( i );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    prev_permutation ( v1.begin( ), v1.end( ), mod_lesser );

    cout << "After the first prev_permutation, vector v1 is:\n v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= 5 ) {
        prev_permutation ( v1.begin( ), v1.end( ), mod_lesser );
        cout << "After another prev_permutation of vector v1,\n v1 =   ( " ;
        for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ;Iter1 ++ )
```

```
            cout << *Iter1 << " ";
        cout << ")." << endl;
        iii++;
    }
}
```

Output

```
The original deque of CInts is deq1 = ( CInt( 1 ), CInt( 5 ), CInt( 10 ) ).
The lexicographically previous permutation doesn't exist
and the lexicographically smallest permutation
has replaced the original ordering of the sequence in deq1.
After one application of prev_permutation,
deq1 = ( CInt( 10 ), CInt( 5 ), CInt( 1 ) ).

Vector v1 is ( -3 -2 -1 0 1 2 3 ).
After the first prev_permutation, vector v1 is:
v1 = ( -3 -2 0 3 2 1 -1 ).
After another prev_permutation of vector v1,
v1 =   ( -3 -2 0 3 -1 2 1 ).
After another prev_permutation of vector v1,
v1 =   ( -3 -2 0 3 -1 1 2 ).
After another prev_permutation of vector v1,
v1 =   ( -3 -2 0 2 3 1 -1 ).
After another prev_permutation of vector v1,
v1 =   ( -3 -2 0 2 -1 3 1 ).
After another prev_permutation of vector v1,
v1 =   ( -3 -2 0 2 -1 1 3 ).
```

push_heap

Adds an element that is at the end of a range to an existing heap consisting of the prior elements in the range.

C++

```
template<class RandomAccessIterator>
void push_heap(
    RandomAccessIterator first,
    RandomAccessIterator last );

template<class RandomAccessIterator, class BinaryPredicate>
void push_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    BinaryPredicate pred);
```

Parameters

first

A random-access iterator addressing the position of the first element in the heap.

last

A random-access iterator addressing the position one past the final element in the range to be converted into a heap.

pred

User-defined predicate function object that defines sense in which one element is less than another. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The element must first be pushed back to the end of an existing heap and then the algorithm is used to add this element to the existing heap.

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

Heaps are an ideal way to implement priority queues and they're used in the implementation of the C++ Standard Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The range excluding the newly added element at the end must be a heap.

The complexity is logarithmic, requiring at most `log(last - first)` comparisons.

Example

C++

```
// alg_push_heap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main() {
    using namespace std;
```

```
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 1 ; i <= 9 ; i++ )
        v1.push_back( i );

    random_shuffle( v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Make v1 a heap with default less than ordering
    make_heap ( v1.begin( ), v1.end( ) );
    cout << "The heaped version of vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Add an element to the heap
    v1.push_back( 10 );
    cout << "The heap v1 with 10 pushed back is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    push_heap( v1.begin( ), v1.end( ) );
    cout << "The reheaped v1 with 10 added is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl << endl;

    // Make v1 a heap with greater than ordering
    make_heap ( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater-than heaped version of v1 is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    v1.push_back(0);
    cout << "The greater-than heap v1 with 11 pushed back is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    push_heap( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "The greater than reheaped v1 with 11 added is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 5 4 8 9 1 6 3 2 7 ).
The heaped version of vector v1 is ( 9 7 8 5 1 6 3 2 4 ).
The heap v1 with 10 pushed back is ( 9 7 8 5 1 6 3 2 4 10 ).
The reheaped v1 with 10 added is ( 10 9 8 5 7 6 3 2 4 1 ).

The greater-than heaped version of v1 is
 ( 1 2 3 4 7 6 8 5 10 9 ).
The greater-than heap v1 with 11 pushed back is
 ( 1 2 3 4 7 6 8 5 10 9 0 ).
The greater than reheaped v1 with 11 added is
 ( 0 1 3 4 2 6 8 5 10 9 7 ).
```

`random_shuffle`

The `std::random_shuffle()` function is deprecated, replaced by std::shuffle. For a code example and more information, see <random> and the Stack Overflow post Why are std::random_shuffle methods being deprecated in C++14?.

`remove`

Eliminates a specified value from a given range without disturbing the order of the remaining elements. Returns the end of a new range free of the specified value.

C++

```
template<class ForwardIterator, class Type>
ForwardIterator remove(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class Type>
ForwardIterator remove(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the range from which elements are being removed.

last

A forward iterator addressing the position one past the final element in the range from which elements are being removed.

value

The value that is to be removed from the range.

Return value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality.

The list class has a more efficient member function version of `remove`, which also relinks pointers.

Example

C++

```
// alg_remove.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
```

```
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value of 7
    new_end = remove ( v1.begin( ), v1.end( ), 7 );

    cout << "Vector v1 with value 7 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To change the sequence size, use erase
    v1.erase (new_end, v1.end( ) );

    cout << "Vector v1 resized with value 7 removed is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
Vector v1 with value 7 removed is ( 4 0 5 1 6 9 3 8 2 9 3 7 8 2 ).
Vector v1 resized with value 7 removed is ( 4 0 5 1 6 9 3 8 2 ).
```

remove_copy

Copies elements from a source range to a destination range, except that elements of a specified value aren't copied, without disturbing the order of the remaining elements. Returns the end of a new destination range.

C++

```
template<class InputIterator, class OutputIterator, class Type>
OutputIterator remove_copy(
```

```
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type>
ForwardIterator2 remove_copy(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    const Type& value);
```

Parameters

exec

The execution policy to use.

first

An input iterator addressing the position of the first element in the range from which elements are being removed.

last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

value

The value that is to be removed from the range.

Return value

A forward iterator addressing the new end position of the destination range, one past the final element of the copy of the remnant sequence free of the specified value.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality and at most (`last` - `first`) assignments.

Example

C++

```
// alg_remove_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2(10);
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle (v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:     ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value of 7
    new_end = remove_copy ( v1.begin( ), v1.end( ), v2.begin( ), 7 );

    cout << "Vector v1 is left unchanged as ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is a copy of v1 with the value 7 removed:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
```

```
        cout << *Iter2 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:     ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
Vector v1 is left unchanged as ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
Vector v2 is a copy of v1 with the value 7 removed:
 ( 4 0 5 1 6 9 3 8 2 0 ).
```

`remove_copy_if`

Copies elements from a source range to a destination range, except for elements that satisfy a predicate. Elements are copied without disturbing the order of the remaining elements. Returns the end of a new destination range.

C++

```
template<class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator remove_copy_if(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class UnaryPredicate>
ForwardIterator2 remove_copy_if(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

An input iterator addressing the position of the first element in the range from which elements are being removed.

last

An input iterator addressing the position one past the final element in the range from which elements are being removed.

result

An output iterator addressing the position of the first element in the destination range to which elements are being removed.

pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return value

A forward iterator addressing the new end position of the destination range, one past the final element of the remnant sequence free of the elements satisfying the predicate.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

There must be enough space in the destination range to contain the remnant elements that will be copied after elements of the specified value are removed.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality and at most (`last` - `first`) assignments.

For information on how these functions behave, see Checked Iterators.

Example

C++

```
// alg_remove_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>
```

```
bool greater6 ( int value ) {
    return value > 6;
}

int main()
{
    using namespace std;
    vector<int> v1, v2(10);
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "The original vector v1 is:      ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements with a value greater than 6
    new_end = remove_copy_if ( v1.begin( ), v1.end( ),
        v2.begin( ), greater6 );

    cout << "After the appliation of remove_copy_if to v1,\n "
         << "vector v1 is left unchanged as ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is a copy of v1 with values greater "
         << "than 6 removed:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != new_end ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:      ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
After the appliation of remove_copy_if to v1,
 vector v1 is left unchanged as ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
Vector v2 is a copy of v1 with values greater than 6 removed:
 ( 4 0 5 1 6 3 2 ).
```

remove_if

Eliminates elements that satisfy a predicate from a given range without disturbing the order of the remaining elements. Returns the end of a new range free of the specified value.

```
C++
```

```
template<class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if(
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator pointing to the position of the first element in the range from which elements are being removed.

last

A forward iterator pointing to the position one past the final element in the range from which elements are being removed.

pred

The unary predicate that must be satisfied is the value of an element is to be replaced.

Return value

A forward iterator addressing the new end position of the modified range, one past the final element of the remnant sequence free of the specified value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not removed remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality.

List has a more efficient member function version of remove which relinks pointers.

Example

C++

```
// alg_remove_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6 ( int value )
{
    return value > 6;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2, new_end;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );
    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Remove elements satisfying predicate greater6
    new_end = remove_if (v1.begin( ), v1.end( ), greater6 );

    cout << "Vector v1 with elements satisfying greater6 removed is\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
```

```
    // To change the sequence size, use erase
    v1.erase (new_end, v1.end( ) );

    cout << "Vector v1 resized elements satisfying greater6 removed is\n ( "
;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
Vector v1 with elements satisfying greater6 removed is
 ( 4 0 5 1 6 3 2 1 6 9 3 7 8 2 ).
Vector v1 resized elements satisfying greater6 removed is
 ( 4 0 5 1 6 3 2 ).
```

replace

Examines each element in a range and replaces it if it matches a specified value.

C++

```
template<class ForwardIterator, class Type>
void replace(
    ForwardIterator first,
    ForwardIterator last,
    const Type& oldVal,
    const Type& newVal);

template<class ExecutionPolicy, class ForwardIterator, class Type>
void replace(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    const Type& oldVal,
    const Type& newVal);
```

Parameters

exec

The execution policy to use.

first

A forward iterator pointing to the position of the first element in the range from which

elements are being replaced.

last
A forward iterator pointing to the position one past the final element in the range from which elements are being replaced.

oldVal
The old value of the elements being replaced.

newVal
The new value being assigned to the elements with the old value.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality and at most (`last` - `first`) assignments of new values.

Example

C++

```
// alg_replace.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    std::vector<int> v;

    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }

    for (int i = 0; i < 3; i++)
    {
        v.push_back(7);
```

```
    }

    std::cout << "The original vector v is:\n ( ";
    for (auto iter = v.begin(); iter != v.end(); iter++)
    {
        std::cout << *iter << " ";
    }
    std::cout << ")." << std::endl;

    // Replace elements with a value of 7 with a value of 700
    replace(v.begin(), v.end(), 7, 700);

    std::cout << "The vector v with 7s replaced with 700s:\n ( ";
    for (auto iter = v.begin(); iter != v.end(); iter++)
    {
        std::cout << *iter << " ";
    }
    std::cout << ")." << std::endl;
}
```

Output

```
The original vector v is:
 ( 0 1 2 3 4 5 6 7 8 9 7 7 7 ).
The vector v with 7s replaced with 700s:
 ( 0 1 2 3 4 5 6 700 8 9 700 700 700 ).
```

replace_copy

Examines each element in a source range and replaces it if it matches a specified value while copying the result into a new destination range.

C++

```
template<class InputIterator, class OutputIterator, class Type>
OutputIterator replace_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    const Type& oldVal,
    const Type& newVal);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type>
ForwardIterator2 replace_copy(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
```

```
    const Type& oldVal,
    const Type& newVal);
```

Parameters

exec
The execution policy to use.

first
An input iterator pointing to the position of the first element in the range from which elements are being replaced.

last
An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

result
An output iterator pointing to the first element in the destination range to where the altered sequence of elements is being copied.

oldVal
The old value of the elements being replaced.

newVal
The new value being assigned to the elements with the old value.

Return value

An output iterator pointing to the position one past the final element in the destination range the altered sequence of elements is copied to.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last - first`) comparisons for equality and at most (`last - first`) assignments of new values.

Example

C++

```
// alg_replace_copy.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> theVector;
    list<int> theList(15);

    for (int i = 0; i <= 9; i++)
    {
        theVector.push_back(i);
    }

    for (int i = 0; i <= 3; i++)
    {
        theVector.push_back(7);
    }

    random_shuffle(theVector.begin(), theVector.end());

    for (int i = 0; i <= 15; i++)
    {
        theVector.push_back(1);
    }

    cout << "The shuffled vector:\n ( ";
    for (auto iter = theVector.begin(); iter != theVector.end(); iter++)
    {
        cout << *iter << " ";
    }
    cout << ")." << endl;

    // Replace the 7s in part of the vector with 70s
    // and copy into another part of the vector
    replace_copy(theVector.begin(), theVector.begin() + 14, theVector.end()
- 15, 7, 70);

    cout << "The vector with instances of 7 replaced with 70 starting at
position 14:\n ( ";
    for (auto iter = theVector.begin(); iter != theVector.end(); iter++)
```

```
    {
        cout << *iter << " ";
    }
    cout << ")." << endl;

    // Replace 7s found in the first 14 positions in the vector with 1s and
then copy the result into a list
    replace_copy(theVector.begin(), theVector.begin() + 14, theList.begin(),
7, 1);

    cout << "List containing the contents of the vector but 7s replaced with
1s.\n ( ";
    for (auto iter = theList.begin(); iter != theList.end(); iter++)
    {
        cout << *iter << " ";
    }
    cout << ")." << endl;
}
```

Because of the `random_shuffle()` call in the preceding code, your output may be different.

Output

```
The shuffled vector:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ).
The vector with instances of 7 replaced with 70 starting at position 14:
 ( 7 1 9 2 0 7 7 3 4 6 8 5 7 7 1 70 1 9 2 0 70 70 3 4 6 8 5 70 70 1 ).
List containing the contents of the vector but 7s replaced with 1s.
 ( 1 1 9 2 0 1 1 3 4 6 8 5 1 1 0 ).
```

`replace_copy_if`

Examines each element in a source range and replaces it if it satisfies a specified predicate while copying the result into a new destination range.

C++

```
template<class InputIterator, class OutputIterator, class UnaryPredicate,
class Type>
OutputIterator replace_copy_if(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    UnaryPredicate pred,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class UnaryPredicate, class Type>
```

```
ForwardIterator2 replace_copy_if(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    UnaryPredicate pred,
    const Type& value);
```

Parameters

exec
The execution policy to use.

first
An input iterator pointing to the position of the first element in the range from which elements are being replaced.

last
An input iterator pointing to the position one past the final element in the range from which elements are being replaced.

result
An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

pred
The unary predicate that must be satisfied is the value of an element is to be replaced.

value
The new value being assigned to the elements whose old value satisfies the predicate.

Return value

An output iterator pointing to the position one past the final element in the destination range the altered sequence of elements is copied to.

Remarks

The source and destination ranges referenced must not overlap and must both be valid: all pointers must be dereferenceable and within the sequences the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality and at most (`last` - `first`) assignments of new values.

Example

C++

```
// alg_replace_copy_if.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

bool greater6 ( int value )
{
    return value > 6;
}

int main()
{
    using namespace std;
    vector<int> v1;
    list<int> L1 (13);
    vector<int>::iterator Iter1;
    list<int>::iterator L_Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
        v1.push_back( 7 );

    random_shuffle ( v1.begin( ), v1.end( ) );

    int iii;
    for ( iii = 0 ; iii <= 13 ; iii++ )
        v1.push_back( 1 );

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements with a value of 7 in the 1st half of a vector
    // with a value of 70 and copy it into the 2nd half of the vector
```

```
    replace_copy_if ( v1.begin( ), v1.begin( ) + 14,v1.end( ) -14,
        greater6 , 70);

    cout << "The vector v1 with values of 70 replacing those greater"
        << "\n than 6 in the 1st half & copied into the 2nd half is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Replace elements in a vector with a value of 70
    // with a value of 1 and copy into a list
    replace_copy_if ( v1.begin( ), v1.begin( ) + 13,L1.begin( ),
        greater6 , -1 );

    cout << "A list copy of vector v1 with the value -1\n replacing "
        << "those greater than 6 is:\n ( " ;
    for ( L_Iter1 = L1.begin( ) ; L_Iter1 != L1.end( ) ; L_Iter1++ )
        cout << *L_Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
 ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
The vector v1 with values of 70 replacing those greater
 than 6 in the 1st half & copied into the 2nd half is:
 ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
A list copy of vector v1 with the value -1
 replacing those greater than 6 is:
 ( 4 7 7 7 0 5 7 1 6 9 3 7 8 2 ).
```

replace_if

Examines each element in a range and replaces it if it satisfies a specified predicate.

C++

```
template<class ForwardIterator, class UnaryPredicate, class Type>
void replace_if(
    ForwardIterator first,
    ForwardIterator last,
    UnaryPredicate pred,
    const Type& value);

template<class ExecutionPolicy, class ForwardIterator, class UnaryPredicate,
class Type>
void replace_if(
    ExecutionPolicy&& exec,
    ForwardIterator first,
```

```
    ForwardIterator last,
    UnaryPredicate pred,
    const Type& value);
```

Parameters

exec
The execution policy to use.

first
A forward iterator pointing to the position of the first element in the range from which elements are being replaced.

last
An iterator pointing to the position one past the final element in the range from which elements are being replaced.

pred
The unary predicate that must be satisfied is the value of an element is to be replaced.

value
The new value being assigned to the elements whose old value satisfies the predicate.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The order of the elements not replaced remains stable.

The algorithm `replace_if` is a generalization of the algorithm `replace`, allowing any predicate to be specified, rather than equality to a specified constant value.

The `operator==` used to determine the equality between elements must impose an equivalence relation between its operands.

The complexity is linear. It makes (`last` - `first`) comparisons for equality and at most (`last` - `first`) assignments of new values.

Example

```
C++
```

```cpp
// alg_replace_if.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater6(int value)
{
    return value > 6;
}

int main()
{
    using namespace std;
    vector<int> v;

    for (int i = 0; i <= 10; i++)
    {
        v.push_back(i);
    }

    cout << "The original vector v:\n ( ";
    for (auto iter = v.begin(); iter != v.end(); iter++)
    {
        cout << *iter << " ";
    }
    cout << ")." << endl;

    // Replace elements satisfying the predicate greater6
    // with a value of 70
    replace_if(v.begin(), v.end(), greater6, 70);

    cout << "The vector v with a value 70 replacing those\n "
        << "elements satisfying the greater6 predicate is:\n ( ";
    for (auto iter = v.begin(); iter != v.end(); iter++)
    {
        cout << *iter << " ";
    }
    cout << ").";
}
```

Output

```
The original vector v:
 ( 0 1 2 3 4 5 6 7 8 9 10 ).
The vector v with a value 70 replacing those
 elements satisfying the greater6 predicate is:
 ( 0 1 2 3 4 5 6 70 70 70 70 ).
```

reverse

Reverses the order of the elements within a range.

```
C++
```

```
template<class BidirectionalIterator>
void reverse(
    BidirectionalIterator first,
    BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(
    ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator last);
```

Parameters

exec

The execution policy to use.

first

A bidirectional iterator pointing to the position of the first element in the range within which the elements are being permuted.

last

A bidirectional iterator pointing to the position one past the final element in the range within which the elements are being permuted.

Remarks

The source range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Example

```
C++
```

```
// alg_reverse.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
```

```
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Reverse the elements in the vector
    reverse (v1.begin( ), v1.end( ) );

    cout << "The modified vector v1 with values reversed is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
( 0 1 2 3 4 5 6 7 8 9 ).
The modified vector v1 with values reversed is:
( 9 8 7 6 5 4 3 2 1 0 ).
```

reverse_copy

Reverses the order of the elements within a source range while copying them into a destination range

C++

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(
    BidirectionalIterator first,
    BidirectionalIterator last,
    OutputIterator result);

template<class ExecutionPolicy, class BidirectionalIterator, class
ForwardIterator>
ForwardIterator reverse_copy(
    ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator last,
    ForwardIterator result);
```

Parameters

exec

The execution policy to use.

first

A bidirectional iterator pointing to the position of the first element in the source range within which the elements are being permuted.

last

A bidirectional iterator pointing to the position one past the final element in the source range within which the elements are being permuted.

result

An output iterator pointing to the position of the first element in the destination range to which elements are being copied.

Return value

An output iterator pointing to the position one past the final element in the destination range the altered sequence of elements is copied to.

Remarks

The source and destination ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Example

C++

```
// alg_reverse_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2( 10 );
```

```
    vector<int>::iterator Iter1, Iter2;

    int i;
    for ( i = 0 ; i <= 9 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "The original vector v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Reverse the elements in the vector
    reverse_copy (v1.begin( ), v1.end( ), v2.begin( ) );

    cout << "The copy v2 of the reversed vector v1 is:\n ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    cout << "The original vector v1 remains unmodified as:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
 ( 0 1 2 3 4 5 6 7 8 9 ).
The copy v2 of the reversed vector v1 is:
 ( 9 8 7 6 5 4 3 2 1 0 ).
The original vector v1 remains unmodified as:
 ( 0 1 2 3 4 5 6 7 8 9 ).
```

rotate

Exchanges the elements in two adjacent ranges.

C++

```
template<class ForwardIterator>
void rotate(
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator rotate(
```

```
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the range to be rotated.

middle

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with the ones in the first part of the range.

last

A forward iterator addressing the position one past the final element in the range to be rotated.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear. It makes at most (`last` - `first`) swaps.

Example

C++

```
// alg_rotate.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main() {
    using namespace std;
    vector<int> v1;
    deque<int> d1;
```

```
    vector<int>::iterator v1Iter1;
    deque<int>::iterator d1Iter1;

    int i;
    for ( i = -3 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii =0 ; ii <= 5 ; ii++ )
    {
        d1.push_back( ii );
    }

    cout << "Vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    rotate ( v1.begin( ), v1.begin( ) + 3 , v1.end( ) );
    cout << "After rotating, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= d1.end( ) - d1.begin( ) ) {
        rotate ( d1.begin( ), d1.begin( ) + 1 , d1.end( ) );
        cout << "After the rotation of a single deque element to the back,\n
d1 is   ( " ;
        for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
            cout << *d1Iter1 << " ";
        cout << ")." << endl;
        iii++;
    }
}
```

Output

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, vector v1 is ( 0 1 2 3 4 5 -3 -2 -1 ).
The original deque d1 is ( 0 1 2 3 4 5 ).
After the rotation of a single deque element to the back,
d1 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
d1 is   ( 2 3 4 5 0 1 ).
After the rotation of a single deque element to the back,
```

```
d1 is   ( 3 4 5 0 1 2 ).
After the rotation of a single deque element to the back,
d1 is   ( 4 5 0 1 2 3 ).
After the rotation of a single deque element to the back,
d1 is   ( 5 0 1 2 3 4 ).
After the rotation of a single deque element to the back,
d1 is   ( 0 1 2 3 4 5 ).
```

`rotate_copy`

Exchanges the elements in two adjacent ranges within a source range and copies the result to a destination range.

C++

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last,
    OutputIterator result );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 rotate_copy(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 middle,
    ForwardIterator1 last,
    ForwardIterator2 result);
```

Parameters

`exec`

The execution policy to use.

`first`

A forward iterator addressing the position of the first element in the range to be rotated.

`middle`

A forward iterator defining the boundary within the range that addresses the position of the first element in the second part of the range whose elements are to be exchanged with the ones in the first part of the range.

last

A forward iterator addressing the position one past the final element in the range to be rotated.

result

An output iterator addressing the position of the first element in the destination range.

Return value

An output iterator addressing the position one past the final element in the destination range.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is linear. It makes at most (`last` - `first`) swaps.

Example

C++

```
// alg_rotate_copy.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1 , v2 ( 9 );
    deque<int> d1 , d2 ( 6 );
    vector<int>::iterator v1Iter , v2Iter;
    deque<int>::iterator d1Iter , d2Iter;

    int i;
    for ( i = -3 ; i <= 5 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii =0 ; ii <= 5 ; ii++ )
        d1.push_back( ii );

    cout << "Vector v1 is ( " ;
```

```
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    rotate_copy ( v1.begin( ), v1.begin( ) + 3 , v1.end( ), v2.begin( ) );
    cout << "After rotating, the vector v1 remains unchanged as:\n v1 = ( "
;
    for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ;v1Iter ++ )
        cout << *v1Iter << " ";
    cout << ")." << endl;

    cout << "After rotating, the copy of vector v1 in v2 is:\n v2 = ( " ;
    for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ;v2Iter ++ )
        cout << *v2Iter << " ";
    cout << ")." << endl;

    cout << "The original deque d1 is ( " ;
    for ( d1Iter = d1.begin( ) ; d1Iter != d1.end( ) ;d1Iter ++ )
        cout << *d1Iter << " ";
    cout << ")." << endl;

    int iii = 1;
    while ( iii <= d1.end( ) - d1.begin( ) )
    {
        rotate_copy ( d1.begin( ), d1.begin( ) + iii , d1.end( ), d2.begin(
) );
        cout << "After the rotation of a single deque element to the back,\n
d2 is   ( " ;
        for ( d2Iter = d2.begin( ) ; d2Iter != d2.end( ) ;d2Iter ++ )
            cout << *d2Iter << " ";
        cout << ")." << endl;
        iii++;
    }
}
```

Output

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, the vector v1 remains unchanged as:
 v1 = ( -3 -2 -1 0 1 2 3 4 5 ).
After rotating, the copy of vector v1 in v2 is:
 v2 = ( 0 1 2 3 4 5 -3 -2 -1 ).
The original deque d1 is ( 0 1 2 3 4 5 ).
After the rotation of a single deque element to the back,
 d2 is   ( 1 2 3 4 5 0 ).
After the rotation of a single deque element to the back,
 d2 is   ( 2 3 4 5 0 1 ).
After the rotation of a single deque element to the back,
 d2 is   ( 3 4 5 0 1 2 ).
After the rotation of a single deque element to the back,
 d2 is   ( 4 5 0 1 2 3 ).
After the rotation of a single deque element to the back,
 d2 is   ( 5 0 1 2 3 4 ).
```

```
After the rotation of a single deque element to the back,
 d2 is   ( 0 1 2 3 4 5 ).
```

sample

C++

```
template<class PopulationIterator, class SampleIterator, class Distance,
class UniformRandomBitGenerator>
SampleIterator sample(
    PopulationIterator first,
    PopulationIterator last,
    SampleIterator out,
    Distance n,
    UniformRandomBitGenerator&& g);
```

search

Searches for the first occurrence of a sequence within a target range whose elements are equal to the ones in a given sequence of elements or whose elements are equivalent in a sense specified by a binary predicate to the elements in the given sequence.

C++

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
ForwardIterator1 search(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator1 search(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class BinaryPredicate>
ForwardIterator1 search(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    BinaryPredicate pred);

template <class ForwardIterator, class Searcher>
ForwardIterator search(
    ForwardIterator first,
    ForwardIterator last,
    const Searcher& searcher);
```

Parameters

exec

The execution policy to use.

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the final element in the range to be searched.

first2

A forward iterator addressing the position of the first element in the range to be matched.

last2

A forward iterator addressing the position one past the final element in the range to be matched.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

searcher

The searcher that encapsulates the pattern to look for and the search algorithm to use.

For more information on searchers, see default_searcher class, boyer_moore_horspool_searcher class, and boyer_moore_searcher class.

Return value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation.

Average complexity is linear with respect to the size of the searched range. Worst case complexity is also linear with respect to the size of the sequence being searched for.

Example

C++

```
// alg_search.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is twice the first
bool twice (int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    list<int> L1;
    vector<int>::iterator Iter1, Iter2;
    list<int>::iterator L1_Iter, L1_inIter;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
```

```
    {
        v1.push_back( 5 * i );
    }
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    int ii;
    for ( ii = 4 ; ii <= 5 ; ii++ )
    {
        L1.push_back( 5 * ii );
    }

    int iii;
    for ( iii = 2 ; iii <= 4 ; iii++ )
    {
        v2.push_back( 10 * iii );
    }

    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    cout << "List L1 = ( " ;
    for ( L1_Iter = L1.begin( ) ; L1_Iter!= L1.end( ) ; L1_Iter++ )
        cout << *L1_Iter << " ";
    cout << ")" << endl;

    cout << "Vector v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
        cout << ")" << endl;

    // Searching v1 for first match to L1 under identity
    vector<int>::iterator result1;
    result1 = search (v1.begin( ), v1.end( ), L1.begin( ), L1.end( ) );

    if ( result1 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
    else
        cout << "There is at least one match of L1 in v1"
            << "\n and the first one begins at "
            << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    result2 = search (v1.begin( ), v1.end( ), v2.begin( ), v2.end( ), twice
);

    if ( result2 == v1.end( ) )
        cout << "There is no match of L1 in v1."
            << endl;
```

```
    else
        cout << "There is a sequence of elements in v1 that "
            << "are equivalent\n to those in v2 under the binary "
            << "predicate twice\n and the first one begins at position "
            << result2 - v1.begin( ) << "." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List L1 = ( 20 25 )
Vector v2 = ( 20 30 40 )
There is at least one match of L1 in v1
and the first one begins at position 4.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.
```

search_n

Searches for the first subsequence in a range that of a specified number of elements having a particular value or a relation to that value as specified by a binary predicate.

C++

```
template<class ForwardIterator1, class Diff2, class Type>
ForwardIterator1 search_n(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    Diff2 count,
    const Type& value);

template<class ForwardIterator1, class Diff2, class Type, class
BinaryPredicate>
ForwardIterator1 search_n(
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    Diff2 count,
    const Type& value,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Size, class
Type>
ForwardIterator search_n(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Size count,
    const Type& value);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Size, class
Type, class BinaryPredicate>
ForwardIterator search_n(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Size count,
    const Type& value,
    BinaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first1

A forward iterator addressing the position of the first element in the range to be searched.

last1

A forward iterator addressing the position one past the final element in the range to be searched.

count

The size of the subsequence being searched for.

value

The value of the elements in the sequence being searched for.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **`true`** when satisfied and **`false`** when not satisfied.

Return value

A forward iterator addressing the position of the first element of the first subsequence that matches the specified sequence or that is equivalent in a sense specified by a binary predicate.

Remarks

The `operator==` used to determine the match between an element and the specified value must impose an equivalence relation between its operands.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Complexity is linear with respect to the size of the searched.

Example

C++

```
// alg_search_n.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

// Return whether second element is 1/2 of the first
bool one_half ( int elem1, int elem2 )
{
    return elem1 == 2 * elem2;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( 5 );
    }

    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 5 * i );
    }

    for ( i = 0 ; i <= 2 ; i++ )
    {
        v1.push_back( 10 );
    }
```

```
    cout << "Vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // Searching v1 for first match to (5 5 5) under identity
    vector<int>::iterator result1;
    result1 = search_n ( v1.begin( ), v1.end( ), 3, 5 );

    if ( result1 == v1.end( ) )
        cout << "There is no match for a sequence ( 5 5 5 ) in v1."
            << endl;
    else
        cout << "There is at least one match of a sequence ( 5 5 5 )"
            << "\n in v1 and the first one begins at "
            << "position "<< result1 - v1.begin( ) << "." << endl;

    // Searching v1 for first match to (5 5 5) under one_half
    vector<int>::iterator result2;
    result2 = search_n (v1.begin( ), v1.end( ), 3, 5, one_half );

    if ( result2 == v1.end( ) )
        cout << "There is no match for a sequence ( 5 5 5 ) in v1"
            << " under the equivalence predicate one_half." << endl;
    else
        cout << "There is a match of a sequence ( 5 5 5 ) "
            << "under the equivalence\n predicate one_half "
            << "in v1 and the first one begins at "
            << "position "<< result2 - v1.begin( ) << "." << endl;
}
```

Output

```
Vector v1 = ( 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 10 10 10 )
There is at least one match of a sequence ( 5 5 5 )
in v1 and the first one begins at position 6.
There is a match of a sequence ( 5 5 5 ) under the equivalence
predicate one_half in v1 and the first one begins at position 15.
```

set_difference

Unites all of the elements that belong to one sorted source range, but not to a second sorted source range, into a single, sorted destination range. A binary predicate may specify the ordering criterion.

C++

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator>
ForwardIterator set_difference(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator set_difference(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the difference of the two source ranges.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the difference of the two source ranges.

pred

User-defined predicate function object that defines the sense in which one element is less than another. The binary predicate takes two arguments and should return **`true`** when the first element is less than the second element and **`false`** otherwise.

Return value

An output iterator addressing the position one past the last element in the sorted destination range representing the difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range shouldn't overlap either of the source ranges and should be large enough to contain the first source range.

The sorted source ranges must each be arranged as a precondition to the application of the `set_difference` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges aren't modified by the algorithm merge.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element such that there are more in the first source range than in the second, then the destination range will contain the number by which the occurrences of those elements in the first source range exceed the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons for nonempty source ranges.

Example

C++

```
// alg_set_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if (elem1 < 0)
        elem1 = - elem1;
    if (elem2 < 0)
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -1 ; i <= 4 ; i++ )
    {
```

```
        v1a.push_back( i );
    }

    int ii;
    for ( ii =-3 ; ii <= 0 ; ii++ )
    {
        v1b.push_back( ii );
    }

    cout << "Original vector v1a with range sorted by the\n "
         << "binary predicate less than is v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
         << "binary predicate less than is v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vectors v2a & v2b with ranges sorted by greater
    vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
    sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
    sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );

    cout << "Original vector v2a with range sorted by the\n "
         << "binary predicate greater is   v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
         << "binary predicate greater is   v2b = ( " ;
    for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
        cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
    vector<int> v3a ( v1a ), v3b ( v1b ) , v3 ( v1 );
    vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
    sort ( v3a.begin( ), v3a.end( ), mod_lesser );
    sort ( v3b.begin( ), v3b.end( ), mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
         << "binary predicate mod_lesser is   v3a = ( " ;
    for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
        cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
         << "binary predicate mod_lesser is   v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
```

```
    cout << ")." << endl;

    // To combine into a difference in asscending
    // order with the default binary predicate less<int>( )
    Result1 = set_difference ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ), v1.begin( ) );
    cout << "Set_difference of source ranges with default order,"
         << "\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To combine into a difference in descending
    // order specify binary predicate greater<int>( )
    Result2 = set_difference ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ),v2.begin( ), greater<int>( ) );
    cout << "Set_difference of source ranges with binary"
         << "predicate greater specified,\n vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // To combine into a difference applying a user
    // defined binary predicate mod_lesser
    Result3 = set_difference ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
    cout << "Set_difference of source ranges with binary "
         << "predicate mod_lesser specified,\n vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
 binary predicate less than is v1b = ( -3 -2 -1 0 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a = ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b = ( 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a = ( 0 -1 1 2 3 4 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b = ( 0 -1 -2 -3 ).
Set_difference of source ranges with default order,
 vector v1mod = ( 1 2 3 4 ).
Set_difference of source ranges with binarypredicate greater specified,
 vector v2mod = ( 4 3 2 1 ).
Set_difference of source ranges with binary predicate mod_lesser specified,
 vector v3mod = ( 1 4 ).
```

`set_intersection`

Unites all of the elements that belong to both sorted source ranges into a single, sorted destination range, where the ordering criterion may be specified by a binary predicate.

C++

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_intersection(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    Compare pred);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator>
ForwardIterator set_intersection(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator set_intersection(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the intersection of the two source ranges.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the intersection of the two source ranges.

pred

User-defined predicate function object that defines the sense in which one element is less than another. The binary predicate takes two arguments and should return **`true`** when the first element is less than the second element and **`false`** otherwise.

Return value

An output iterator addressing the position one past the last element in the sorted destination range representing the intersection of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by

incrementation.

The destination range shouldn't overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the merge algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges aren't modified by the algorithm.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons for nonempty source ranges.

Example

C++

```
// alg_set_intersection.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>   // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
```

```
using namespace std;
vector<int> v1a, v1b, v1 ( 12 );
vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

// Constructing vectors v1a & v1b with default less than ordering
int i;
for ( i = -1 ; i <= 3 ; i++ )
    v1a.push_back( i );

int ii;
for ( ii =-3 ; ii <= 1 ; ii++ )
    v1b.push_back( ii );

cout << "Original vector v1a with range sorted by the\n "
     << "binary predicate less than is v1a = ( " ;
for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
    cout << *Iter1a << " ";
cout << ")." << endl;

cout << "Original vector v1b with range sorted by the\n "
     << "binary predicate less than is v1b = ( " ;
for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
    cout << *Iter1b << " ";
cout << ")." << endl;

// Constructing vectors v2a & v2b with ranges sorted by greater
vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );

cout << "Original vector v2a with range sorted by the\n "
     << "binary predicate greater is   v2a = ( " ;
for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
    cout << *Iter2a << " ";
cout << ")." << endl;

cout << "Original vector v2b with range sorted by the\n "
     << "binary predicate greater is   v2b = ( " ;
for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
    cout << *Iter2b << " ";
cout << ")." << endl;

// Constructing vectors v3a & v3b with ranges sorted by mod_lesser
vector<int> v3a ( v1a ), v3b ( v1b ) , v3 ( v1 );
vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
sort ( v3a.begin( ), v3a.end( ), mod_lesser );
sort ( v3b.begin( ), v3b.end( ), mod_lesser );

cout << "Original vector v3a with range sorted by the\n "
     << "binary predicate mod_lesser is   v3a = ( " ;
for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
    cout << *Iter3a << " ";
cout << ")." << endl;
```

```
    cout << "Original vector v3b with range sorted by the\n "
         << "binary predicate mod_lesser is   v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To combine into an intersection in asscending order with the
    // default binary predicate less<int>( )
    Result1 = set_intersection ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ), v1.begin( ) );
    cout << "Intersection of source ranges with default order,"
         << "\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; ++Iter1 )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To combine into an intersection in descending order, specify
    // binary predicate greater<int>( )
    Result2 = set_intersection ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ),v2.begin( ), greater<int>( ) );
    cout << "Intersection of source ranges with binary predicate"
            << " greater specified,\n vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; ++Iter2 )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // To combine into an intersection applying a user-defined
    // binary predicate mod_lesser
    Result3 = set_intersection ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
    cout << "Intersection of source ranges with binary predicate "
            << "mod_lesser specified,\n vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; ++Iter3 )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( -1 0 1 2 3 ).
Original vector v1b with range sorted by the
 binary predicate less than is v1b = ( -3 -2 -1 0 1 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a = ( 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b = ( 1 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a = ( 0 -1 1 2 3 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b = ( 0 -1 1 -2 -3 ).
Intersection of source ranges with default order,
 vector v1mod = ( -1 0 1 ).
```

```
Intersection of source ranges with binary predicate greater specified,
 vector v2mod = ( 1 0 -1 ).
Intersection of source ranges with binary predicate mod_lesser specified,
 vector v3mod = ( 0 -1 1 2 3 ).
```

set_symmetric_difference

Unites all of the elements that belong to one, but not both, of the sorted source ranges into a single, sorted destination range. A binary predicate may specify the ordering criterion.

C++

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_symmetric_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_symmetric_difference(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator>
ForwardIterator set_symmetric_difference(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator set_symmetric_difference(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result,
    Compare pred);
```

Parameters

`exec`
The execution policy to use.

`first1`
An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

`last1`
An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

`first2`
An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

`last2`
An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the symmetric difference of the two source ranges.

`result`
An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the symmetric difference of the two source ranges.

`pred`
User-defined predicate function object that defines the sense in which one element is less than another. The binary predicate takes two arguments and should return `true` when the first element is less than the second element and `false` otherwise.

Return value

An output iterator addressing the position one past the last element in the sorted destination range representing the symmetric difference of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range shouldn't overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge*` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges aren't modified by the algorithm merge.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the absolute value of the number by which the occurrences of those elements in the one of the source ranges exceeds the occurrences of those elements in the second source range.

The complexity of the algorithm is linear with at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons for nonempty source ranges.

Example

C++

```
// alg_set_sym_diff.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser (int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
```

```
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less-than ordering
    int i;
    for ( i = -1 ; i <= 4 ; i++ )
    {
        v1a.push_back( i );
    }

    int ii;
    for ( ii =-3 ; ii <= 0 ; ii++ )
    {
        v1b.push_back( ii );
    }

    cout << "Original vector v1a with range sorted by the\n "
         << "binary predicate less than is v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
         << "binary predicate less than is v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vectors v2a & v2b with ranges sorted by greater
    vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
    sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
    sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );

    cout << "Original vector v2a with range sorted by the\n "
         << "binary predicate greater is   v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
         << "binary predicate greater is   v2b = ( " ;
    for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
        cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
    vector<int> v3a ( v1a ), v3b ( v1b ) , v3 ( v1 );
```

```
    vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
    sort ( v3a.begin( ), v3a.end( ), mod_lesser );
    sort ( v3b.begin( ), v3b.end( ), mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
         << "binary predicate mod_lesser is   v3a = ( " ;
    for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
        cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
         << "binary predicate mod_lesser is   v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To combine into a symmetric difference in ascending
    // order with the default binary predicate less<int>( )
    Result1 = set_symmetric_difference ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ), v1.begin( ) );
    cout << "Set_symmetric_difference of source ranges with default order,"
         << "\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To combine into a symmetric difference in descending
    // order, specify binary predicate greater<int>( )
    Result2 = set_symmetric_difference ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ),v2.begin( ), greater<int>( ) );
    cout << "Set_symmetric_difference of source ranges with binary"
         << "predicate greater specified,\n vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // To combine into a symmetric difference applying a user
    // defined binary predicate mod_lesser
    Result3 = set_symmetric_difference ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
    cout << "Set_symmetric_difference of source ranges with binary "
         << "predicate mod_lesser specified,\n vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( -1 0 1 2 3 4 ).
Original vector v1b with range sorted by the
 binary predicate less than is v1b = ( -3 -2 -1 0 ).
```

```
Original vector v2a with range sorted by the
 binary predicate greater is   v2a = ( 4 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b = ( 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a = ( 0 -1 1 2 3 4 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b = ( 0 -1 -2 -3 ).
Set_symmetric_difference of source ranges with default order,
 vector v1mod = ( -3 -2 1 2 3 4 ).
Set_symmetric_difference of source ranges with binarypredicate greater
specified,
 vector v2mod = ( 4 3 2 1 -2 -3 ).
Set_symmetric_difference of source ranges with binary predicate mod_lesser
specified,
 vector v3mod = ( 1 4 ).
```

set_union

Unites all of the elements that belong to at least one of two sorted source ranges into a single, sorted destination range. A binary predicate may specify the ordering criterion.

C++

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_union(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result );

template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
OutputIterator set_union(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    InputIterator2 last2,
    OutputIterator result,
    Compare pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator>
ForwardIterator set_union(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator set_union(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator2 last2,
    ForwardIterator result,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

last1

An input iterator addressing the position one past the last element in the first of two sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

first2

An input iterator addressing the position of the first element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

last2

An input iterator addressing the position one past the last element in second of two consecutive sorted source ranges to be united and sorted into a single range representing the union of the two source ranges.

result

An output iterator addressing the position of the first element in the destination range where the two source ranges are to be united into a single sorted range representing the union of the two source ranges.

pred

User-defined predicate function object that defines the sense in which one element is

less than another. The binary predicate takes two arguments and should return `true` when the first element is less than the second element and `false` otherwise.

Return value

An output iterator addressing the position one past the last element in the sorted destination range representing the union of the two source ranges.

Remarks

The sorted source ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation.

The destination range shouldn't overlap either of the source ranges and should be large enough to contain the destination range.

The sorted source ranges must each be arranged as a precondition to the application of the `merge` algorithm in accordance with the same ordering as is to be used by the algorithm to sort the combined ranges.

The operation is stable as the relative order of elements within each range is preserved in the destination range. The source ranges aren't modified by the algorithm `merge`.

The value types of the input iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements. When there are equivalent elements in both source ranges, the elements in the first range precede the elements from the second source range in the destination range. If the source ranges contain duplicates of an element, then the destination range will contain the maximum number of those elements that occur in both source ranges.

The complexity of the algorithm is linear with at most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

Example

C++

```
// alg_set_union.cpp
// compile with: /EHsc
#include <vector>
```

```cpp
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;
    vector<int> v1a, v1b, v1 ( 12 );
    vector<int>::iterator Iter1a, Iter1b, Iter1, Result1;

    // Constructing vectors v1a & v1b with default less than ordering
    int i;
    for ( i = -1 ; i <= 3 ; i++ )
    {
        v1a.push_back( i );
    }

    int ii;
    for ( ii =-3 ; ii <= 1 ; ii++ )
    {
        v1b.push_back( ii );
    }

    cout << "Original vector v1a with range sorted by the\n "
         << "binary predicate less than is v1a = ( " ;
    for ( Iter1a = v1a.begin( ) ; Iter1a != v1a.end( ) ; Iter1a++ )
        cout << *Iter1a << " ";
    cout << ")." << endl;

    cout << "Original vector v1b with range sorted by the\n "
         << "binary predicate less than is v1b = ( " ;
    for ( Iter1b = v1b.begin( ) ; Iter1b != v1b.end( ) ; Iter1b++ )
        cout << *Iter1b << " ";
    cout << ")." << endl;

    // Constructing vectors v2a & v2b with ranges sorted by greater
    vector<int> v2a ( v1a ) , v2b ( v1b ) , v2 ( v1 );
    vector<int>::iterator Iter2a, Iter2b, Iter2, Result2;
    sort ( v2a.begin( ), v2a.end( ), greater<int>( ) );
    sort ( v2b.begin( ), v2b.end( ), greater<int>( ) );

    cout << "Original vector v2a with range sorted by the\n "
         << "binary predicate greater is   v2a = ( " ;
    for ( Iter2a = v2a.begin( ) ; Iter2a != v2a.end( ) ; Iter2a++ )
        cout << *Iter2a << " ";
```

```
    cout << ")." << endl;

    cout << "Original vector v2b with range sorted by the\n "
         << "binary predicate greater is   v2b = ( " ;
    for ( Iter2b = v2b.begin( ) ; Iter2b != v2b.end( ) ; Iter2b++ )
        cout << *Iter2b << " ";
    cout << ")." << endl;

    // Constructing vectors v3a & v3b with ranges sorted by mod_lesser
    vector<int> v3a ( v1a ), v3b ( v1b ) , v3 ( v1 );
    vector<int>::iterator Iter3a, Iter3b, Iter3, Result3;
    sort ( v3a.begin( ), v3a.end( ), mod_lesser );
    sort ( v3b.begin( ), v3b.end( ), mod_lesser );

    cout << "Original vector v3a with range sorted by the\n "
         << "binary predicate mod_lesser is   v3a = ( " ;
    for ( Iter3a = v3a.begin( ) ; Iter3a != v3a.end( ) ; Iter3a++ )
        cout << *Iter3a << " ";
    cout << ")." << endl;

    cout << "Original vector v3b with range sorted by the\n "
         << "binary predicate mod_lesser is   v3b = ( " ;
    for ( Iter3b = v3b.begin( ) ; Iter3b != v3b.end( ) ; Iter3b++ )
        cout << *Iter3b << " ";
    cout << ")." << endl;

    // To combine into a union in ascending order with the default
        // binary predicate less<int>( )
    Result1 = set_union ( v1a.begin( ), v1a.end( ),
        v1b.begin( ), v1b.end( ), v1.begin( ) );
    cout << "Union of source ranges with default order,"
         << "\n vector v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != Result1 ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // To combine into a union in descending order, specify binary
    // predicate greater<int>( )
    Result2 = set_union ( v2a.begin( ), v2a.end( ),
        v2b.begin( ), v2b.end( ),v2.begin( ), greater<int>( ) );
    cout << "Union of source ranges with binary predicate greater "
         << "specified,\n vector v2mod = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != Result2 ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // To combine into a union applying a user-defined
    // binary predicate mod_lesser
    Result3 = set_union ( v3a.begin( ), v3a.end( ),
        v3b.begin( ), v3b.end( ), v3.begin( ), mod_lesser );
    cout << "Union of source ranges with binary predicate "
         << "mod_lesser specified,\n vector v3mod = ( " ; ;
    for ( Iter3 = v3.begin( ) ; Iter3 != Result3 ; Iter3++ )
        cout << *Iter3 << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
Original vector v1a with range sorted by the
 binary predicate less than is v1a = ( -1 0 1 2 3 ).
Original vector v1b with range sorted by the
 binary predicate less than is v1b = ( -3 -2 -1 0 1 ).
Original vector v2a with range sorted by the
 binary predicate greater is   v2a = ( 3 2 1 0 -1 ).
Original vector v2b with range sorted by the
 binary predicate greater is   v2b = ( 1 0 -1 -2 -3 ).
Original vector v3a with range sorted by the
 binary predicate mod_lesser is   v3a = ( 0 -1 1 2 3 ).
Original vector v3b with range sorted by the
 binary predicate mod_lesser is   v3b = ( 0 -1 1 -2 -3 ).
Union of source ranges with default order,
 vector v1mod = ( -3 -2 -1 0 1 2 3 ).
Union of source ranges with binary predicate greater specified,
 vector v2mod = ( 3 2 1 0 -1 -2 -3 ).
Union of source ranges with binary predicate mod_lesser specified,
 vector v3mod = ( 0 -1 1 2 3 ).
```

`shuffle`

Shuffles (rearranges) elements for a given range by using a random number generator.

C++

```
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void shuffle(
    RandomAccessIterator first,
    RandomAccessIterator last,
    UniformRandomNumberGenerator&& gen);
```

Parameters

`first`

An iterator to the first element in the range to be shuffled, inclusive. Must meet the requirements of `RandomAccessIterator` and `ValueSwappable`.

`last`

An iterator to the last element in the range to be shuffled, exclusive. Must meet the requirements of `RandomAccessIterator` and `ValueSwappable`.

gen

The random number generator that the `shuffle()` function will use for the operation. Must meet the requirements of a `UniformRandomNumberGenerator`.

Remarks

For more information, and a code sample that uses `shuffle()`, see <random>.

`sort`

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

C++

```
template<class RandomAccessIterator>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);

template<class ExecutionPolicy, class RandomAccessIterator>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A random-access iterator addressing the position of the first element in the range to be

sorted.

last
A random-access iterator addressing the position one past the final element in the range to be sorted.

pred
User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. This binary predicate takes two arguments and returns **true** if the two arguments are in order and **false** otherwise. This comparator function must impose a strict weak ordering on pairs of elements from the sequence. For more information, see Algorithms.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm isn't stable and so doesn't guarantee that the relative ordering of equivalent elements will be preserved. The algorithm `stable_sort` does preserve this original ordering.

The average of a sort complexity is `O( N log N )`, where *N* = *last* - *first*.

Example

C++

```
// alg_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector<int> v1;
```

```
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    int ii;
    for ( ii = 0 ; ii <= 5 ; ii++ )
    {
        v1.push_back( 2 * ii + 1 );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order. specify binary predicate
    sort( v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted (greater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    sort( v1.begin( ), v1.end( ), UDgreater );
    cout << "Resorted (UDgreater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
Resorted (UDgreater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
```

sort_heap

Converts a heap into a sorted range.

C++

```
template<class RandomAccessIterator>
void sort_heap(
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

first

A random-access iterator addressing the position of the first element in the target heap.

last

A random-access iterator addressing the position one past the final element in the target heap.

pred

User-defined predicate function object that defines the sense in which one element is less than another. A comparison predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

Heaps have two properties:

- The first element is always the largest.
- Elements may be added or removed in logarithmic time.

After the application if this algorithm, the range it was applied to is no longer a heap.

`sort_heap` isn't a stable sort because the relative order of equivalent elements isn't necessarily preserved.

Heaps are an ideal way to implement priority queues and they're used in the implementation of the C++ Standard Library container adaptor priority_queue Class.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

The complexity is at most `N log N`, where *N* = *`last`* - *`first`*.

Example

C++

```
// alg_sort_heap.cpp
// compile with: /EHsc
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>
#include <string>
#include <vector>
using namespace std;

void print(const string& s, const vector<int>& v)
{
    cout << s << ": ( ";

    for (auto i = v.begin(); i != v.end(); ++i)
    {
        cout << *i << " ";
    }

    cout << ")" << endl;
}

int main()
{
    vector<int> v;
    for (int i = 1; i <= 9; ++i)
    {
        v.push_back(i);
    }
    print("Initially", v);

    random_shuffle(v.begin(), v.end());
    print("After random_shuffle", v);

    make_heap(v.begin(), v.end());
    print("     After make_heap", v);

    sort_heap(v.begin(), v.end());
    print("     After sort_heap", v);

    random_shuffle(v.begin(), v.end());
    print("             After random_shuffle", v);

    make_heap(v.begin(), v.end(), greater<int>());
    print("After make_heap with greater<int>", v);

    sort_heap(v.begin(), v.end(), greater<int>());
```

```
    print("After sort_heap with greater<int>", v);
}
```

Output

```
Initially: ( 1 2 3 4 5 6 7 8 9 )
After random_shuffle: ( 5 4 8 9 1 6 3 2 7 )
     After make_heap: ( 9 7 8 5 1 6 3 2 4 )
     After sort_heap: ( 1 2 3 4 5 6 7 8 9 )
             After random_shuffle: ( 1 3 6 8 9 5 4 2 7 )
After make_heap with greater<int>: ( 1 2 4 3 9 5 6 8 7 )
After sort_heap with greater<int>: ( 9 8 7 6 5 4 3 2 1 )
```

`stable_partition`

Classifies elements in a range into two disjoint sets, with the elements that satisfy a unary predicate preceding the ones that fail to satisfy it, preserving the relative order of equivalent elements.

C++

```
template<class BidirectionalIterator, class UnaryPredicate>
BidirectionalIterator stable_partition(
    BidirectionalIterator first,
    BidirectionalIterator last,
    UnaryPredicate pred );

template<class ExecutionPolicy, class BidirectionalIterator, class
UnaryPredicate>
BidirectionalIterator stable_partition(
    ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator last,
    UnaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

A bidirectional iterator addressing the position of the first element in the range to be partitioned.

last

A bidirectional iterator addressing the position one past the final element in the range to be partitioned.

pred

User-defined predicate function object that defines the condition to be satisfied if an element is to be classified. A unary predicate takes a single argument and returns **true** if satisfied, or **false** if not satisfied.

Return value

A bidirectional iterator addressing the position of the first element in the range to not satisfy the predicate condition.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements *a* and *b* are equivalent, but not necessarily equal, if both `pred( a, b )` is false and `pred( b, a )` is false, where *pred* is the parameter-specified predicate. The `stable_partition` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved. The algorithm `partition` doesn't necessarily preserve this original ordering.

Example

C++

```
// alg_stable_partition.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

bool greater5 ( int value )
{
    return value > 5;
}

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2, result;
```

```
    int i;
    for ( i = 0 ; i <= 10 ; i++ )
        v1.push_back( i );

    int ii;
    for ( ii = 0 ; ii <= 4 ; ii++ )
        v1.push_back( 5 );

    random_shuffle(v1.begin( ), v1.end( ) );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Partition the range with predicate greater10
    result = stable_partition (v1.begin( ), v1.end( ), greater5 );
    cout << "The partitioned set of elements in v1 is:\n ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "The first element in v1 to fail to satisfy the"
         << "\n predicate greater5 is: " << *result << "." << endl;
}
```

Output

```
Vector v1 is ( 4 10 5 5 5 5 5 1 6 9 3 7 8 2 0 5 ).
The partitioned set of elements in v1 is:
 ( 10 6 9 7 8 4 5 5 5 5 5 1 3 2 0 5 ).
The first element in v1 to fail to satisfy the
 predicate greater5 is: 4.
```

stable_sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate. It preserves the relative ordering of equivalent elements.

C++

```
template<class BidirectionalIterator>
void stable_sort(
    BidirectionalIterator first,
    BidirectionalIterator last );

template<class BidirectionalIterator, class Compare>
```

```
void stable_sort(
    BidirectionalIterator first,
    BidirectionalIterator last,
    Compare pred );

template<class ExecutionPolicy, class RandomAccessIterator>
void stable_sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void stable_sort(
    ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare pred);
```

Parameters

exec

The execution policy to use.

first

A bidirectional iterator addressing the position of the first element in the range to be sorted.

last

A bidirectional iterator addressing the position one past the final element in the range to be sorted.

pred

User-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Remarks

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation.

Elements are equivalent, but not necessarily equal, if neither is less than the other. The `sort` algorithm is stable and guarantees that the relative ordering of equivalent elements will be preserved.

The run-time complexity of `stable_sort` depends on the amount of memory available, but the best case (given sufficient memory) is `O(N log N)` and the worst case is `O(N (log N)^2)`, where *N* = *`last`* - *`first`*. Usually, the `sort` algorithm is faster than `stable_sort`.

Example

C++

```
// alg_stable_sort.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // For greater<int>( )
#include <iostream>

// Return whether first element is greater than the second
bool UDgreater (int elem1, int elem2 )
{
    return elem1 > elem2;
}

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( 2 * i );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    stable_sort(v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // To sort in descending order, specify binary predicate
    stable_sort(v1.begin( ), v1.end( ), greater<int>( ) );
    cout << "Resorted (greater) vector v1 = ( " ;
```

```
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;

    // A user-defined (UD) binary predicate can also be used
    stable_sort(v1.begin( ), v1.end( ), UDgreater );
    cout << "Resorted (UDgreater) vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 0 2 4 6 8 10 0 2 4 6 8 10 )
Sorted vector v1 = ( 0 0 2 2 4 4 6 6 8 8 10 10 )
Resorted (greater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
Resorted (UDgreater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
```

swap

The first override exchanges the values of two objects. The second override exchanges the values between two arrays of objects.

C++

```
template<class Type>
void swap(
    Type& left,
    Type& right);
template<class Type, size_t N>
void swap(
    Type (& left)[N],
    Type (& right)[N]);
```

Parameters

left

For the first override, the first object to have its contents exchanged. For the second override, the first array of objects to have its contents exchanged.

right

For the first override, the second object to have its contents exchanged. For the second override, the second array of objects to have its contents exchanged.

Remarks

The first overload is designed to operate on individual objects. The second overload swaps the contents of objects between two arrays.

Example

C++

```
// alg_swap.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2;
    vector<int>::iterator Iter1, Iter2, result;

    for ( int i = 0 ; i <= 10 ; i++ )
    {
        v1.push_back( i );
    }

    for ( int ii = 0 ; ii <= 4 ; ii++ )
    {
        v2.push_back( 5 );
    }

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    swap( v1, v2 );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    cout << "Vector v2 is ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 0 1 2 3 4 5 6 7 8 9 10 ).
Vector v2 is ( 5 5 5 5 5 ).
Vector v1 is ( 5 5 5 5 5 ).
Vector v2 is ( 0 1 2 3 4 5 6 7 8 9 10 ).
```

swap_ranges

Exchanges the elements of one range with the elements of another, equal sized range.

C++

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
   ForwardIterator1 first1,
   ForwardIterator1 last1,
   ForwardIterator2 first2 );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 swap_ranges(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2);
```

Parameters

exec

The execution policy to use.

first1

A forward iterator pointing to the first position of the first range whose elements are to be exchanged.

last1

A forward iterator pointing to one past the final position of the first range whose elements are to be exchanged.

first2

A forward iterator pointing to the first position of the second range whose elements are

to be exchanged.

Return value

A forward iterator pointing to one past the final position of the second range whose elements are to be exchanged.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position is reachable from the first by incrementation. The second range has to be as large as the first range.

The complexity is linear with *last1* - *first1* swaps performed. If elements from containers of the same type are being swapped, them the `swap` member function from that container should be used, because the member function typically has constant complexity.

Example

C++

```
// alg_swap_ranges.cpp
// compile with: /EHsc
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1;
    deque<int> d1;
    vector<int>::iterator v1Iter1;
    deque<int>::iterator d1Iter1;

    int i;
    for ( i = 0 ; i <= 5 ; i++ )
    {
        v1.push_back( i );
    }

    int ii;
    for ( ii =4 ; ii <= 9 ; ii++ )
    {
        d1.push_back( 6 );
```

```
    }

    cout << "Vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "Deque d1 is  ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;

    swap_ranges ( v1.begin( ), v1.end( ), d1.begin( ) );

    cout << "After the swap_range, vector v1 is ( " ;
    for ( v1Iter1 = v1.begin( ) ; v1Iter1 != v1.end( ) ;v1Iter1 ++ )
        cout << *v1Iter1 << " ";
    cout << ")." << endl;

    cout << "After the swap_range deque d1 is   ( " ;
    for ( d1Iter1 = d1.begin( ) ; d1Iter1 != d1.end( ) ;d1Iter1 ++ )
        cout << *d1Iter1 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 0 1 2 3 4 5 ).
Deque d1 is  ( 6 6 6 6 6 6 ).
After the swap_range, vector v1 is ( 6 6 6 6 6 6 ).
After the swap_range deque d1 is   ( 0 1 2 3 4 5 ).
```

`transform`

Applies a specified function object to each element in a source range or to a pair of elements from two source ranges. Then, it copies the return values of the function object into a destination range.

C++

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(
    InputIterator first1,
    InputIterator last1,
    OutputIterator result,
    UnaryFunction func );

template<class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryFunction>
```

```
OutputIterator transform(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    OutputIterator result,
    BinaryFunction func );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class UnaryOperation>
ForwardIterator2 transform(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    UnaryOperation op);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class ForwardIterator, class BinaryOperation>
ForwardIterator transform(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    ForwardIterator result,
    BinaryOperation binary_op);
```

Parameters

exec

The execution policy to use.

first1

An input iterator addressing the position of the first element in the first source range to operate on.

last1

An input iterator addressing the position one past the final element in the first source range to operate on.

first2

An input iterator addressing the position of the first element in the second source range to operate on.

result

An output iterator addressing the position of the first element in the destination range.

func

User-defined unary function object used in the first version of the algorithm to apply to

each element in the first source range or a user-defined (UD) binary function object used in the second version of the algorithm that is applied pairwise, in a forward order, to the two source ranges.

Return value

An output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.

Remarks

The ranges referenced must be valid; all pointers must be dereferenceable and within each sequence the last position must be reachable from the first by incrementation. The destination range must be large enough to contain the transformed source range.

If *result* is set equal to `first1` in the first version of the algorithm, then the source and destination ranges will be the same and the sequence will be modified in place. But the `result` may not address a position within the range [`first1` + 1, `last1`).

The complexity is linear. It makes at most (`last1` - `first1`) comparisons.

Example

C++

```
// alg_transform.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
    Type Factor;   // The value to multiply by
public:
    // Constructor initializes the value to multiply by
    MultValue ( const Type& value ) : Factor ( value ) { }

    // The function call for the element to be multiplied
    Type operator( ) ( Type& elem ) const
    {
        return elem * Factor;
    }
```

```
};

int main()
{
    using namespace std;
    vector<int> v1, v2 ( 7 ), v3 ( 7 );
    vector<int>::iterator Iter1, Iter2 , Iter3;

    // Constructing vector v1
    int i;
    for ( i = -4 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Modifying the vector v1 in place
    transform (v1.begin( ), v1.end( ), v1.begin( ), MultValue<int> ( 2 ) );
    cout << "The elements of the vector v1 multiplied by 2 in place gives:"
            << "\n v1mod = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Using transform to multiply each element by a factor of 5
    transform ( v1.begin( ), v1.end( ), v2.begin( ), MultValue<int> ( 5 ) );

    cout << "Multiplying the elements of the vector v1mod\n "
            << "by the factor 5 & copying to v2 gives:\n v2 = ( " ;
    for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
        cout << *Iter2 << " ";
    cout << ")." << endl;

    // The second version of transform used to multiply the
    // elements of the vectors v1mod & v2 pairwise
    transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin( ),
        multiplies<int>( ) );

    cout << "Multiplying elements of the vectors v1mod and v2 pairwise "
            << "gives:\n v3 = ( " ;
    for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
        cout << *Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Original vector v1 = ( -4 -3 -2 -1 0 1 2 ).
The elements of the vector v1 multiplied by 2 in place gives:
```

```
v1mod = ( -8 -6 -4 -2 0 2 4 ).
Multiplying the elements of the vector v1mod
by the factor 5 & copying to v2 gives:
v2 = ( -40 -30 -20 -10 0 10 20 ).
Multiplying elements of the vectors v1mod and v2 pairwise gives:
v3 = ( 320 180 80 20 0 20 80 ).
```

`unique`

Removes duplicate elements that are next to each other in a specified range.

C++

```
template<class ForwardIterator>
ForwardIterator unique(
    ForwardIterator first,
    ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator unique(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class
BinaryPredicate>
ForwardIterator unique(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    BinaryPredicate pred);
```

Parameters

`exec`

The execution policy to use.

`first`

A forward iterator addressing the position of the first element in the range to be scanned for duplicate removal.

last

A forward iterator addressing the position one past the final element in the range to be scanned for duplicate removal.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

A forward iterator to the new end of the modified sequence that contains no consecutive duplicates, addressing the position one past the last element not removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements isn't changed.

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation. The number of elements in the sequence isn't changed by the algorithm `unique` and the elements beyond the end of the modified sequence are dereferenceable but not specified.

The complexity is linear, requiring `(last - first) - 1` comparisons.

List provides a more efficient member function "unique", which may perform better.

These algorithms can't be used on an associative container.

Example

C++

```
// alg_unique.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>
```

```
using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 == elem2;
};

int main()
{
    vector<int> v1;
    vector<int>::iterator v1_Iter1, v1_Iter2, v1_Iter3,
            v1_NewEnd1, v1_NewEnd2, v1_NewEnd3;

    int i;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        v1.push_back( 5 );
        v1.push_back( -5 );
    }

    int ii;
    for ( ii = 0 ; ii <= 3 ; ii++ )
    {
        v1.push_back( 4 );
    }
    v1.push_back( 7 );

    cout << "Vector v1 is ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    // Remove consecutive duplicates
    v1_NewEnd1 = unique ( v1.begin( ), v1.end( ) );

    cout << "Removing adjacent duplicates from vector v1 gives\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    // Remove consecutive duplicates under the binary prediate mod_equals
    v1_NewEnd2 = unique ( v1.begin( ), v1_NewEnd1 , mod_equal );

    cout << "Removing adjacent duplicates from vector v1 under the\n "
            << " binary predicate mod_equal gives\n ( " ;
    for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
        cout << *v1_Iter2 << " ";
    cout << ")." << endl;
```

```
    // Remove elements if preceded by an element that was greater
    v1_NewEnd3 = unique ( v1.begin( ), v1_NewEnd2, greater<int>( ) );

    cout << "Removing adjacent elements satisfying the binary\n "
            << " predicate greater<int> from vector v1 gives ( " ;
    for ( v1_Iter3 = v1.begin( ) ; v1_Iter3 != v1_NewEnd3 ; v1_Iter3++ )
        cout << *v1_Iter3 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is ( 5 -5 5 -5 5 -5 5 -5 4 4 4 4 7 ).
Removing adjacent duplicates from vector v1 gives
( 5 -5 5 -5 5 -5 5 -5 4 7 ).
Removing adjacent duplicates from vector v1 under the
  binary predicate mod_equal gives
( 5 4 7 ).
Removing adjacent elements satisfying the binary
  predicate greater<int> from vector v1 gives ( 5 7 ).
```

unique_copy

Copies elements from a source range into a destination range except for the duplicate elements that are next to each other.

C++

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator result );

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryPredicate pred );

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 unique_copy(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class
```

```
    ForwardIterator2,
    class BinaryPredicate>
    ForwardIterator2 unique_copy(ExecutionPolicy&& exec,
        ForwardIterator1 first,
        ForwardIterator1 last,
        ForwardIterator2 result,
        BinaryPredicate pred);
```

Parameters

exec

The execution policy to use.

first

A forward iterator addressing the position of the first element in the source range to be copied.

last

A forward iterator addressing the position one past the final element in the source range to be copied.

result

An output iterator addressing the position of the first element in the destination range that is receiving the copy with consecutive duplicates removed.

pred

User-defined predicate function object that defines the condition to be satisfied if two elements are to be taken as equivalent. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied.

Return value

An output iterator addressing the position one past the final element in the destination range that is receiving the copy with consecutive duplicates removed.

Remarks

Both forms of the algorithm remove the second duplicate of a consecutive pair of equal elements.

The operation of the algorithm is stable so that the relative order of the undeleted elements isn't changed.

The ranges referenced must be valid; all pointers must be dereferenceable and within a sequence the last position is reachable from the first by incrementation.

The complexity is linear, requiring (`last` - `first`) comparisons.

Example

C++

```
// alg_unique_copy.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>
#include <ostream>

using namespace std;

// Return whether modulus of elem1 is equal to modulus of elem2
bool mod_equal ( int elem1, int elem2 ) {
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 == elem2;
};

int main() {
    vector<int> v1;
    vector<int>::iterator v1_Iter1, v1_Iter2,
            v1_NewEnd1, v1_NewEnd2;

    int i;
    for ( i = 0 ; i <= 1 ; i++ ) {
        v1.push_back( 5 );
        v1.push_back( -5 );
    }

    int ii;
    for ( ii = 0 ; ii <= 2 ; ii++ )
        v1.push_back( 4 );
    v1.push_back( 7 );

    int iii;
    for ( iii = 0 ; iii <= 5 ; iii++ )
        v1.push_back( 10 );

    cout << "Vector v1 is\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1.end( ) ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;
```

```
    // Copy first half to second, removing consecutive duplicates
    v1_NewEnd1 = unique_copy ( v1.begin( ), v1.begin( ) + 8, v1.begin( ) + 8
);

    cout << "Copying the first half of the vector to the second half\n "
            << "while removing adjacent duplicates gives\n ( " ;
    for ( v1_Iter1 = v1.begin( ) ; v1_Iter1 != v1_NewEnd1 ; v1_Iter1++ )
        cout << *v1_Iter1 << " ";
    cout << ")." << endl;

    int iv;
    for ( iv = 0 ; iv <= 7 ; iv++ )
        v1.push_back( 10 );

    // Remove consecutive duplicates under the binary prediate mod_equals
    v1_NewEnd2 = unique_copy ( v1.begin( ), v1.begin( ) + 14,
        v1.begin( ) + 14 , mod_equal );

    cout << "Copying the first half of the vector to the second half\n "
            << " removing adjacent duplicates under mod_equals gives\n ( " ;
    for ( v1_Iter2 = v1.begin( ) ; v1_Iter2 != v1_NewEnd2 ; v1_Iter2++ )
        cout << *v1_Iter2 << " ";
    cout << ")." << endl;
}
```

Output

```
Vector v1 is
 ( 5 -5 5 -5 4 4 4 7 10 10 10 10 10 10 ).
Copying the first half of the vector to the second half
 while removing adjacent duplicates gives
 ( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 ).
Copying the first half of the vector to the second half
  removing adjacent duplicates under mod_equals gives
 ( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 5 4 7 5 4 7 ).
```

upper_bound

Finds the position of the first element in an ordered range that has a value that's greater than a specified value. A binary predicate specifies the ordering criterion.

C++

```
template<class ForwardIterator, class Type>
ForwardIterator upper_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value);
```

```
template<class ForwardIterator, class Type, class Compare>
ForwardIterator upper_bound(
    ForwardIterator first,
    ForwardIterator last,
    const Type& value,
    Compare pred);
```

Parameters

first

The position of the first element in the range to be searched.

last

The position one past the final element in the range to be searched.

value

The value in the ordered range that needs to be exceeded by the value of the element addressed by the iterator returned.

pred

User-defined comparison predicate function object that defines the sense in which one element is less than another. A comparison predicate takes two arguments and returns **`true`** when satisfied and **`false`** when not satisfied.

Return value

A forward iterator to the position of the first element that has a value greater than a specified value.

Remarks

The sorted source range referenced must be valid; all iterators must be dereferenceable and within the sequence the last position must be reachable from the first by incrementation.

A sorted range is a precondition of the use of `upper_bound` and where the ordering criterion is the same as specified by the comparison predicate.

The range isn't modified by `upper_bound`.

The value types of the forward iterators must be less-than comparable to be ordered. That is, given two elements, you can determine either that one is less than the other, or

that they're equivalent. (Here, equivalent means that neither is less than the other.) This comparison results in an ordering between the nonequivalent elements.

The complexity of the algorithm is logarithmic for random-access iterators and linear otherwise, with the number of steps proportional to (`last - first`).

Example

C++

```
// alg_upper_bound.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>      // greater<int>( )
#include <iostream>

// Return whether modulus of elem1 is less than modulus of elem2
bool mod_lesser( int elem1, int elem2 )
{
    if ( elem1 < 0 )
        elem1 = - elem1;
    if ( elem2 < 0 )
        elem2 = - elem2;
    return elem1 < elem2;
}

int main()
{
    using namespace std;

    vector<int> v1;
    // Constructing vector v1 with default less-than ordering
    for ( auto i = -1 ; i <= 4 ; ++i )
    {
        v1.push_back( i );
    }

    for ( auto ii =-3 ; ii <= 0 ; ++ii )
    {
        v1.push_back( ii );
    }

    cout << "Starting vector v1 = ( " ;
    for (const auto &Iter : v1)
        cout << Iter << " ";
    cout << ")." << endl;

    sort(v1.begin(), v1.end());
    cout << "Original vector v1 with range sorted by the\n "
        << "binary predicate less than is v1 = ( " ;
    for (const auto &Iter : v1)
```

```
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vector v2 with range sorted by greater
    vector<int> v2(v1);

    sort(v2.begin(), v2.end(), greater<int>());

    cout << "Original vector v2 with range sorted by the\n "
        << "binary predicate greater is v2 = ( " ;
    for (const auto &Iter : v2)
        cout << Iter << " ";
    cout << ")." << endl;

    // Constructing vectors v3 with range sorted by mod_lesser
    vector<int> v3(v1);
    sort(v3.begin(), v3.end(), mod_lesser);

    cout << "Original vector v3 with range sorted by the\n "
        << "binary predicate mod_lesser is v3 = ( " ;
    for (const auto &Iter : v3)
        cout << Iter << " ";
    cout << ")." << endl;

    // Demonstrate upper_bound

    vector<int>::iterator Result;

    // upper_bound of 3 in v1 with default binary predicate less<int>()
    Result = upper_bound(v1.begin(), v1.end(), 3);
    cout << "The upper_bound in v1 for the element with a value of 3 is: "
        << *Result << "." << endl;

    // upper_bound of 3 in v2 with the binary predicate greater<int>( )
    Result = upper_bound(v2.begin(), v2.end(), 3, greater<int>());
    cout << "The upper_bound in v2 for the element with a value of 3 is: "
        << *Result << "." << endl;

    // upper_bound of 3 in v3 with the binary predicate mod_lesser
    Result = upper_bound(v3.begin(), v3.end(), 3, mod_lesser);
    cout << "The upper_bound in v3 for the element with a value of 3 is: "
        << *Result << "." << endl;
}
```

Output

```
Starting vector v1 = ( -1 0 1 2 3 4 -3 -2 -1 0 ).
Original vector v1 with range sorted by the
 binary predicate less than is v1 = ( -3 -2 -1 -1 0 0 1 2 3 4 ).
Original vector v2 with range sorted by the
 binary predicate greater is v2 = ( 4 3 2 1 0 0 -1 -1 -2 -3 ).
Original vector v3 with range sorted by the
 binary predicate mod_lesser is v3 = ( 0 0 -1 -1 1 -2 2 -3 3 4 ).
```

```
The upper_bound in v1 for the element with a value of 3 is: 4.
The upper_bound in v2 for the element with a value of 3 is: 2.
The upper_bound in v3 for the element with a value of 3 is: 4.
```

<allocators>

Article • 12/07/2021

Defines several templates that help allocate and free memory blocks for node-based containers.

Syntax

```
C++

#include <allocators>
```

> **Note**
>
> <allocators> is deprecated, starting with Visual Studio 2019 version 16.3.

Remarks

The <allocators> header provides six allocator templates that can be used to select memory-management strategies for node-based containers. For use with these templates, it also provides several different synchronization filters to tailor the memory-management strategy to a variety of different multithreading schemes (including none). You can speed up your app, or reduce its memory requirements, by matching a memory management strategy to its memory usage patterns and synchronization requirements.

The allocator templates are implemented with reusable components that can be customized or replaced to provide additional memory-management strategies.

The node-based containers in the C++ Standard Library (std::list, std::set, std::multiset, std::map, and std::multimap) store their elements in individual nodes. All the nodes for a particular container type are the same size, so the flexibility of a general-purpose memory manager is not needed. Because the size of each memory block is known at compile time, the memory manager can be much simpler and faster.

When used with containers that are not node-based (such as the C++ Standard Library containers std::vector std::deque, and std::basic_string), the allocator templates will work correctly, but are not likely to provide any performance improvement over the default allocator.

An allocator is a class template that describes an object that manages storage allocation and freeing for objects and arrays of objects of a designated type. Allocator objects are used by several container class templates in the C++ Standard Library.

The allocators are all templates of this type:

```
C++

template<class Type>
class allocator;
```

where the template argument `Type` is the type managed by the allocator instance. The C++ Standard Library provides a default allocator, class template allocator, which is defined in <memory>. The <allocators> header provides the following allocators:

- allocator_newdel
- allocator_unbounded
- allocator_fixed_size
- allocator_variable_size
- allocator_suballoc
- allocator_chunklist

Use an appropriate instantiation of an allocator as the second type argument when creating a container, such as the following code example.

```
C++

#include <list>
#include <allocators>
std::list<int, stdext::allocators::allocator_chunklist<int> > _List0;
```

_List0 allocates nodes with `allocator_chunklist` and the default synchronization filter.

Use the macro ALLOCATOR_DECL to create allocator templates with synchronization filters other than the default:

```
C++

#include <list>
#include <allocators>
ALLOCATOR_DECL(CACHE_CHUNKLIST, stdext::allocators::sync_per_thread, Alloc);
std::list<int, alloc<int> > _List1;
```

_Lst1 allocates nodes with `allocator_chunklist` and the sync_per_thread synchronization filter.

A block allocator is a cache or a filter. A cache is a class template that takes one argument of type std::size_t. It defines a block allocator that allocates and deallocates memory blocks of a single size. It must obtain memory using operator `new`, but it need not make a separate call to operator `new` for each block. It may, for example, suballocate from a larger block or cache deallocated blocks for subsequent reallocation.

With a compiler that cannot rebind the value of the std::size_t argument used when the template was instantiated is not necessarily the value of the argument _Sz passed to a cache's member functions allocate and deallocate.

<allocators> provides the following cache templates:

- cache_freelist
- cache_suballoc
- cache_chunklist

A filter is a block allocator that implements its member functions using another block allocator, which is passed to it as a template argument. The most common form of filter is a synchronization filter, which applies a synchronization policy to control access to the member functions of an instance of another block allocator. <allocators> provides the following synchronization filters:

- sync_none
- sync_per_container
- sync_per_thread
- sync_shared

<allocators> also provides the filter rts_alloc, which holds multiple block allocator instances and determines which instance to use for allocation or deallocation at runtime instead of at compile time. It is used with compilers that cannot compile rebind.

A synchronization policy determines how an allocator instance handles simultaneous allocation and deallocation requests from multiple threads. The simplest policy is to pass all requests directly through to the underlying cache object, leaving synchronization management to the user. A more complex policy could be to use a mutex to serialize access to the underlying cache object.

If a compiler supports compiling both single-threaded and multi-threaded applications, the default synchronization filter for single-threaded applications is `sync_none`; for all other cases it is `sync_shared`.

The cache template `cache_freelist` takes a max class argument, which determines the maximum number of elements to be stored in the free list.

<allocators> provides the following max classes:

- max_none
- max_unbounded
- max_fixed_size
- max_variable_size

Macros

Macro	Description
ALLOCATOR_DECL	Yields an allocator class template.
CACHE_CHUNKLIST	Yields `stdext::allocators::cache_chunklist<sizeof(Type)>`.
CACHE_FREELIST	Yields `stdext::allocators::cache_freelist<sizeof(Type), max>`.
CACHE_SUBALLOC	Yields `stdext::allocators::cache_suballoc<sizeof(Type)>`.
SYNC_DEFAULT	Yields a synchronization filter.

Operators

Operator	Description
operator!= (<allocators>)	Tests for inequality between allocator objects of a specified class.
operator== (<allocators>)	Tests for equality between allocator objects of a specified class.

Classes

Class	Description
allocator_base	Defines the base class and common functions needed to create a user-defined allocator from a synchronization filter.

Class	Description
allocator_chunklist	Describes an object that manages storage allocation and freeing for objects using a cache of type cache_chunklist.
allocator_fixed_size	Describes an object that manages storage allocation and freeing for objects of type `Type` using a cache of type cache_freelist with a length managed by max_fixed_size.
allocator_newdel	Implements an allocator that uses **operator delete** to deallocate a memory block and **operator new** to allocate a memory block.
allocator_suballoc	Describes an object that manages storage allocation and freeing for objects of type `Type` using a cache of type cache_suballoc.
allocator_unbounded	Describes an object that manages storage allocation and freeing for objects of type `Type` using a cache of type cache_freelist with a length managed by max_unbounded.
allocator_variable_size	Describes an object that manages storage allocation and freeing for objects of type `Type` using a cache of type cache_freelist with a length managed by max_variable_size.
cache_chunklist	Defines a block allocator that allocates and deallocates memory blocks of a single size.
cache_freelist	Defines a block allocator that allocates and deallocates memory blocks of a single size.
cache_suballoc	Defines a block allocator that allocates and deallocates memory blocks of a single size.
freelist	Manages a list of memory blocks.
max_fixed_size	Describes a max class object that limits a freelist object to a fixed maximum length.
max_none	Describes a max class object that limits a freelist object to a maximum length of zero.
max_unbounded	Describes a max class object that does not limit the maximum length of a freelist object.
max_variable_size	Describes a max class object that limits a freelist object to a maximum length that is roughly proportional to the number of allocated memory blocks.
rts_alloc	The rts_alloc class template describes a filter that holds an array of cache instances and determines which instance to use for allocation and deallocation at runtime instead of at compile time.

Class	Description
sync_none	Describes a synchronization filter that provides no synchronization.
sync_per_container	Describes a synchronization filter that provides a separate cache object for each allocator object.
sync_per_thread	Describes a synchronization filter that provides a separate cache object for each thread.
sync_shared	Describes a synchronization filter that uses a mutex to control access to a cache object that is shared by all allocators.

Requirements

Header: <allocators>

Namespace: stdext

See also

Header Files Reference

`<allocators>` macros

Article • 12/07/2021

ALLOCATOR_DECL
CACHE_CHUNKLIST

CACHE_FREELIST

CACHE_SUBALLOC

SYNC_DEFAULT

ALLOCATOR_DECL

Yields an allocator class template.

```
C++

#define ALLOCATOR_DECL(cache, sync, name) <alloc_template>
```

Remarks

The macro yields a template definition `template <class Type> class name {.....}` and a specialization `template <> class name<void> {.....}` which together define an allocator class template that uses the synchronization filter `sync` and a cache of type `cache`.

For compilers that can compile rebind, the resulting template definition looks like this:

```
C++

struct rebind
   {    /* convert a name<Type> to a name<Other> */
   typedef name<Other> other;
   };
```

For compilers that cannot compile rebind the resulting template definition looks like this:

```
C++

template <class Type<class name
    : public stdext::allocators::allocator_base<Type,
```

```
    sync<stdext::allocators::rts_alloc<cache>>>
{
public:
    name() {}
    template <class Other>
    name(const name<Other>&) {}
    template <class Other>
    name& operator= (const name<Other>&)
    {
        return *this;
    }
};
```

CACHE_CHUNKLIST

Yields `stdext::allocators::cache_chunklist<sizeof(Type)>`.

C++

```
#define CACHE_CHUNKLIST <cache_class>
```

Remarks

CACHE_FREELIST

Yields `stdext::allocators::cache_freelist<sizeof(Type), max>`.

C++

```
#define CACHE_FREELIST(max) <cache_class>
```

Remarks

CACHE_SUBALLOC

Yields `stdext::allocators::cache_suballoc<sizeof(Type)>`.

C++

```
#define CACHE_SUBALLOC <cache_class>
```

Remarks

SYNC_DEFAULT

Yields a synchronization filter.

```
C++
```

```
#define SYNC_DEFAULT <sync_template>
```

Remarks

If a compiler supports compiling both single-threaded and multi-threaded applications, for single-threaded applications the macro yields `stdext::allocators::sync_none`; in all other cases it yields `stdext::allocators::sync_shared`.

See also

<allocators>

`<allocators>` operators

Article • 12/07/2021

These are the global template operator functions defined in `<allocators>`. For class member operator functions, see the class documentation.

operator!=
operator==

operator!=

Tests for inequality between allocator objects of a specified class.

```
C++
```

```
template <class Type, class Sync>
bool operator!=(
    const allocator_base<Type, Sync>& left,
    const allocator_base<Type, Sync>& right);
```

Parameters

left
One of the allocator objects to be tested for inequality.

right
One of the allocator objects to be tested for inequality.

Return Value

`true` if the allocator objects are not equal; `false` if allocator objects are equal.

Remarks

The template operator returns `!(left == right)`.

operator==

Tests for equality between allocator objects of a specified class.

C++

```
template <class Type, class Sync>
bool operator==(
    const allocator_base<Type, Sync>& left,
    const allocator_base<Type, Sync>& right);
```

Parameters

left
One of the allocator objects to be tested for equality.

right
One of the allocator objects to be tested for equality.

Return Value

`true` if the allocator objects are equal; `false` if allocator objects are not equal.

Remarks

This template operator returns `left.equals(right)`.

See also

<allocators>

allocator_base Class

Article • 06/09/2022

Defines the base class and common functions needed to create a user-defined allocator from a synchronization filter.

Syntax

```
C++
```

```
template <class Type, class Sync>
class allocator_base
```

Parameters

Type
The type of elements allocated by the allocator.

Sync
The synchronization policy for the allocator, which is sync_none Class, sync_per_container Class, sync_per_thread Class, or sync_shared Class.

Constructors

Constructor	Description
allocator_base	Constructs an object of type `allocator_base`.

Typedefs

Type name	Description
const_pointer	A type that provides a constant pointer to the type of object managed by the allocator.
const_reference	A type that provides a constant reference to type of object managed by the allocator.
difference_type	A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.

Type name	Description
pointer	A type that provides a pointer to the type of object managed by the allocator.
reference	A type that provides a reference to the type of object managed by the allocator.
size_type	An unsigned integral type that can represent the length of any sequence that an object of type `allocator_base` can allocate.
value_type	A type that is managed by the allocator.

Member functions

Member function	Description
_Charalloc	Allocates storage for an array of type `char`.
_Chardealloc	Frees storage for the array containing elements of type `char`.
address	Finds the address of an object whose value is specified.
allocate	Allocates a block of memory large enough to store at least some specified number of elements.
construct	Constructs a specific type of object at a specified address that is initialized with a specified value.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
destroy	Calls an objects destructor without deallocating the memory where the object was stored.
max_size	Returns the number of elements of type *Type* that could be allocated by an object of class allocator before the free memory is used up.

Requirements

Header: <allocators>

Namespace: stdext

allocator_base::_Charalloc

Allocates storage for an array of type `char`.

```
C++
```

```
char *_Charalloc(size_type count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

This member function is used by containers when compiled with a compiler that can't compile rebind. It implements `_Charalloc` for the user-defined allocator by returning the result of a call to the `allocate` function of the synchronization filter.

allocator_base::_Chardealloc

Frees storage for the array containing elements of type `char`.

```
C++
```

```
void _Chardealloc(void* ptr, size_type count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

This member function is used by containers when compiled with a compiler that can't compile rebind. It implements `_Chardealloc` for the user-defined allocator by calling the `deallocate` function of the synchronization filter. The pointer ptr must have been earlier

returned by a call to `_Charalloc` for an allocator object that compares equal to `*this`, allocating an array object of the same size and type. `_Chardealloc` never throws an exception.

allocator_base::address

Finds the address of an object whose value is specified.

```
C++
```

```
pointer address(reference val);

const_pointer address(const_reference val);
```

Parameters

val
The const or nonconst value of the object whose address is being searched for.

Return Value

A const or nonconst pointer to the object found of, respectively, const or nonconst value.

Remarks

This member function is implemented for the user-defined allocator by returning `&val`.

allocator_base::allocate

Allocates a block of memory large enough to store at least some specified number of elements.

```
C++
```

```
template <class Other>
pointer allocate(size_type _Nx, const Other* _Hint = 0);

pointer allocate(size_type _Nx);
```

Parameters

_Nx
The number of elements in the array to be allocated.

_Hint
This parameter is ignored.

Return Value

A pointer to the allocated object.

Remarks

The member function implements memory allocation for the user-defined allocator by returning the result of a call to the `allocate` function of the synchronization filter of type Type `*` if `_Nx == 1`, otherwise by returning the result of a call to `operator new(_Nx * sizeof(Type))` cast to type Type `*`.

allocator_base::allocator_base

Constructs an object of type `allocator_base`.

```
C++
```

```
allocator_base();

template <class Other>
allocator_base(const allocator_base<Other, Sync>& right);
```

Parameters

right
The allocator object to be copied.

Remarks

The first constructor constructs an allocator_base instance. The second constructor constructs an `allocator_base` instance such that for any `allocator_base<Type, _Sync>` instance `a`, `allocator_base<Type, Sync>(allocator_base<Other, Sync>(a)) == a`.

allocator_base::const_pointer

A type that provides a constant pointer to the type of object managed by the allocator.

```
C++

typedef const Type *const_pointer;
```

allocator_base::const_reference

A type that provides a constant reference to type of object managed by the allocator.

```
C++

typedef const Type& const_reference;
```

allocator_base::construct

Constructs a specific type of object at a specified address that is initialized with a specified value.

```
C++

void construct(pointer ptr, const Type& val);
```

Parameters

ptr
A pointer to the location where the object is to be constructed.

val
The value with which the object being constructed is to be initialized.

Remarks

This member function is implemented for the user-defined allocator by calling `new((void*)ptr Type(val)`.

allocator_base::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(pointer ptr, size_type _Nx);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

_Nx
The number of objects to be deallocated from storage.

Remarks

This member function is implemented for the user-defined allocator by calling `deallocate(ptr)` on the synchronization filter `Sync` if `_Nx == 1`, otherwise by calling `operator delete(_Nx * ptr)`.

allocator_base::destroy

Calls an objects destructor without deallocating the memory where the object was stored.

```
C++
```

```
void destroy(pointer ptr);
```

Parameters

ptr
A pointer designating the address of the object to be destroyed.

Remarks

This member function is implemented for the user-defined allocator by calling `ptr->~Type()`.

allocator_base::difference_type

A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.

```
C++
```

```
typedef std::ptrdiff_t difference_type;
```

allocator_base::max_size

Returns the number of elements of type `Type` that could be allocated by an object of class allocator before the free memory is used up.

```
C++
```

```
size_type max_size() const;
```

Return Value

The number of elements that could be allocated.

Remarks

This member function is implemented for the user-defined allocator by returning `(size_t)-1 / sizeof(Type)` if `0 < (size_t)-1 / sizeof(Type)`, otherwise `1`.

allocator_base::pointer

A type that provides a pointer to the type of object managed by the allocator.

```
C++
```

```
typedef Type *pointer;
```

allocator_base::reference

A type that provides a reference to the type of object managed by the allocator.

```
```

```
C++
```

```
typedef Type& reference;
```

allocator_base::size_type

An unsigned integral type that can represent the length of any sequence that an object of type `allocator_base` can allocate.

```
C++
```

```
typedef std::size_t size_type;
```

allocator_base::value_type

A type that is managed by the allocator.

```
C++
```

```
typedef Type value_type;
```

See also

<allocators>

allocator_chunklist Class

Article • 08/03/2021

Describes an object that manages storage allocation and freeing for objects using a cache of type cache_chunklist.

Syntax

```
C++
```

```
template <class Type>
class allocator_chunklist;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_CHUNKLIST, SYNC_DEFAULT, allocator_chunklist);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

allocator_fixed_size Class

Article • 08/03/2021

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type cache_freelist with a length managed by max_fixed_size.

Syntax

```
C++
```

```
template <class Type>
class allocator_fixed_size;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_fixed_size<10>), SYNC_DEFAULT, allocator_fixed_size);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

allocator_newdel Class

Article • 08/03/2021

Implements an allocator that uses **operator delete** to deallocate a memory block and **operator new** to allocate a memory block.

Syntax

```
C++
```

```
template <class Type>
class allocator_newdel;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_FREELIST stdext::allocators::max_none), SYNC_DEFAULT, allocator_newdel);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

allocator_suballoc Class

Article • 08/03/2021

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type cache_suballoc.

Syntax

```
C++
```

```
template <class Type>
class allocator_suballoc;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_SUBALLOC, SYNC_DEFAULT, allocator_suballoc);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

allocator_unbounded Class

Article • 08/03/2021

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type cache_freelist with a length managed by max_unbounded.

Syntax

```
C++
```

```
template <class Type>
class allocator_unbounded;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_unbounded), SYNC_DEFAULT, allocator_unbounded);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

allocator_variable_size Class

Article • 08/03/2021

Describes an object that manages storage allocation and freeing for objects of type *Type* using a cache of type cache_freelist with a length managed by max_variable_size.

Syntax

```
C++
```

```
template <class Type>
class allocator_variable_size;
```

Parameters

Type
The type of elements allocated by the allocator.

Remarks

The ALLOCATOR_DECL macro passes this class as the *name* parameter in the following statement: `ALLOCATOR_DECL(CACHE_FREELIST(stdext::allocators::max_variable_size), SYNC_DEFAULT, allocator_variable_size);`

Requirements

Header: <allocators>

Namespace: stdext

See also

<allocators>

cache_chunklist Class

Article • 08/03/2021

Defines a block allocator that allocates and deallocates memory blocks of a single size.

Syntax

```
C++
```

```
template <std::size_t Sz, std::size_t Nelts = 20>
class cache_chunklist
```

Parameters

Sz
The number of elements in the array to be allocated.

Remarks

This class template uses **operator new** to allocate chunks of raw memory, suballocating blocks to allocate storage for a memory block when needed; it stores deallocated memory blocks in a separate free list for each chunk, and uses **operator delete** to deallocate a chunk when none of its memory blocks is in use.

Each memory block holds *Sz* bytes of usable memory and a pointer to the chunk that it belongs to. Each chunk holds `Nelts` memory blocks, three pointers, an int and the data that **operator new** and **operator delete** require.

Constructors

Constructor	Description
cache_chunklist	Constructs an object of type `cache_chunklist`.

Member functions

Member function	Description

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_chunklist::allocate

Allocates a block of memory.

```
C++

void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_chunklist::cache_chunklist

Constructs an object of type `cache_chunklist`.

```
C++

cache_chunklist();
```

Remarks

cache_chunklist::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

See also

<allocators>

cache_freelist Class

Article • 08/03/2021

Defines a block allocator that allocates and deallocates memory blocks of a single size.

Syntax

```
C++
```

```
template <std::size_t Sz, class Max>
class cache_freelist
```

Parameters

Sz
The number of elements in the array to be allocated.

Max
The max class representing the maximum size of the free list. This can be max_fixed_size, max_none, max_unbounded, or max_variable_size.

Remarks

The cache_freelist class template maintains a free list of memory blocks of size *Sz*. When the free list is full it uses **operator delete** to deallocate memory blocks. When the free list is empty it uses **operator new** to allocate new memory blocks. The maximum size of the free list is determined by the class max class passed in the *Max* parameter.

Each memory block holds *Sz* bytes of usable memory and the data that **operator new** and **operator delete** require.

Constructors

Constructor	Description
cache_freelist	Constructs an object of type `cache_freelist`.

Member functions

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_freelist::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_freelist::cache_freelist

Constructs an object of type `cache_freelist`.

```
C++
```

```
cache_freelist();
```

Remarks

cache_freelist::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

See also

<allocators>

cache_suballoc Class

Article • 08/03/2021

Defines a block allocator that allocates and deallocates memory blocks of a single size.

Syntax

```
C++
```

```
template <std::size_t Sz, size_t Nelts = 20>
class cache_suballoc
```

Parameters

Sz
The number of elements in the array to be allocated.

Remarks

The cache_suballoc class template stores deallocated memory blocks in a free list with unbounded length, using `freelist<sizeof(Type), max_unbounded>`, and suballocates memory blocks from a larger chunk allocated with **operator new** when the free list is empty.

Each chunk holds `Sz * Nelts` bytes of usable memory and the data that **operator new** and **operator delete** require. Allocated chunks are never freed.

Constructors

Constructor	Description
cache_suballoc	Constructs an object of type `cache_suballoc`.

Member functions

Member function	Description
allocate	Allocates a block of memory.

Member function	Description
deallocate	Frees a specified number of objects from storage beginning at a specified position.

Requirements

Header: <allocators>

Namespace: stdext

cache_suballoc::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

cache_suballoc::cache_suballoc

Constructs an object of type `cache_suballoc`.

```
C++
```

```
cache_suballoc();
```

Remarks

cache_suballoc::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

See also

<allocators>

freelist Class

Article • 08/03/2021

Manages a list of memory blocks.

Syntax

```
C++
```

```
template <std::size_t Sz, class Max>
class freelist : public Max
```

Parameters

Sz
The number of elements in the array to be allocated.

Max
The max class representing the maximum number of elements to be stored in the free list. The max class can be max_none, max_unbounded, max_fixed_size, or max_variable_size.

Remarks

This class template manages a list of memory blocks of size *Sz* with the maximum length of the list determined by the max class passed in *Max*.

Constructors

Constructor	Description
freelist	Constructs an object of type `freelist`.

Member functions

Member function	Description
pop	Removes the first memory block from the free list.

Member function	Description
push	Adds a memory block to the list.

Requirements

Header: <allocators>

Namespace: stdext

freelist::freelist

Constructs an object of type `freelist`.

```
C++

freelist();
```

Remarks

freelist::pop

Removes the first memory block from the free list.

```
C++

void *pop();
```

Return Value

Returns a pointer to the memory block removed from the list.

Remarks

The member function returns NULL if the list is empty. Otherwise, it removes the first memory block from the list.

freelist::push

Adds a memory block to the list.

```
C++
```

```
bool push(void* ptr);
```

Parameters

ptr
A pointer to the memory block to be added to the free list.

Return Value

`true` if the `full` function of the max class returns `false`; otherwise, the `push` function returns `false`.

Remarks

If the `full` function of the max class returns `false`, this member function adds the memory block pointed to by *ptr* to the head of the list.

See also

<allocators>

max_fixed_size Class

Article • 08/03/2021

Describes a max class object that limits a freelist object to a fixed maximum length.

Syntax

```
C++
```

```
template <std::size_t Max>
class max_fixed_size
```

Parameters

Max
The max class that determines the maximum number of elements to store in the `freelist`.

Constructors

Constructor	Description
max_fixed_size	Constructs an object of type `max_fixed_size`.

Member functions

Member function	Description
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_fixed_size::allocated

Increments the count of allocated memory blocks.

```
C++
```

```
void allocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

The member function does nothing. This member function is called after each successful call by `cache_freelist::allocate` to operator **`new`**. The argument *_Nx* is the number of memory blocks in the chunk allocated by operator **`new`**.

max_fixed_size::deallocated

Decrements the count of allocated memory blocks.

```
C++
```

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator **`delete`**. The argument *_Nx* is the number of memory blocks in the chunk deallocated by operator **`delete`**.

max_fixed_size::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
C++

bool full();
```

Return Value

`true` if `Max <= _Nblocks`; otherwise, **`false`**.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns **`true`**, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator **`delete`** to deallocate the block.

max_fixed_size::max_fixed_size

Constructs an object of type `max_fixed_size`.

```
C++

max_fixed_size();
```

Remarks

This constructor initializes the stored value `_Nblocks` to zero.

max_fixed_size::released

Decrements the count of memory blocks on the free list.

```
C++

void released();
```

Remarks

Decrements the stored value `_Nblocks`. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_fixed_size::saved

Increments the count of memory blocks on the free list.

```
C++

void saved();
```

Remarks

This member function increments the stored value `_Nblocks`. This member function is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

<allocators>

max_none Class

Article • 08/03/2021

Describes a max class object that limits a freelist object to a maximum length of zero.

Syntax

```
C++
```

```
template <std::size_t Max>
class max_none
```

Parameters

Max
The max class that determines the maximum number of elements to store in the `freelist`.

Member functions

Member function	Description
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_none::allocated

Increments the count of allocated memory blocks.

```
C++
```

```
void allocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

This member function does nothing. It is called after each successful call by `cache_freelist::allocate` to operator `new`. The argument *_Nx* is the number of memory blocks in the chunk allocated by operator `new`.

max_none::deallocated

Decrements the count of allocated memory blocks.

```
C++
```

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator `delete`. The argument *_Nx* is the number of memory blocks in the chunk deallocated by operator `delete`.

max_none::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
C++
```

```
bool full();
```

Return Value

This member function always returns `true`.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns `true`, `deallocate` puts the memory block on the free list; if it returns `false`, `deallocate` calls operator `delete` to deallocate the block.

max_none::released

Decrements the count of memory blocks on the free list.

```
C++
```

```
void released();
```

Remarks

This member function does nothing. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_none::saved

Increments the count of memory blocks on the free list.

```
C++
```

```
void saved();
```

Remarks

This member function does nothing. It is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

<allocators>

max_unbounded Class

Article • 08/03/2021

Describes a max class object that does not limit the maximum length of a freelist object.

Syntax

```
C++

class max_unbounded
```

Member functions

Member function	Description
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_unbounded::allocated

Increments the count of allocated memory blocks.

```
C++

void allocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

This member function does nothing. It is called after each successful call by `cache_freelist::allocate` to operator **`new`**. The argument *_Nx* is the number of memory blocks in the chunk allocated by operator **`new`**.

max_unbounded::deallocated

Decrements the count of allocated memory blocks.

```
C++
```

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

The member function does nothing. This member function is called after each call by `cache_freelist::deallocate` to operator **`delete`**. The argument *_Nx* is the number of memory blocks in the chunk deallocated by operator **`delete`**.

max_unbounded::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
C++
```

```
bool full();
```

Return Value

The member function always returns `false`.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns `true`, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator `delete` to deallocate the block.

max_unbounded::released

Decrements the count of memory blocks on the free list.

```
C++
```

```
void released();
```

Remarks

This member function does nothing. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_unbounded::saved

Increments the count of memory blocks on the free list.

```
C++
```

```
void saved();
```

Remarks

This member function does nothing. It is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

<allocators>

max_variable_size Class

Article • 08/03/2021

Describes a max class object that limits a freelist object to a maximum length that is roughly proportional to the number of allocated memory blocks.

Syntax

```
C++
```

```
class max_variable_size
```

Constructors

Constructor	Description
max_variable_size	Constructs an object of type `max_variable_size`.

Member functions

Member function	Description
allocated	Increments the count of allocated memory blocks.
deallocated	Decrements the count of allocated memory blocks.
full	Returns a value that specifies whether more memory blocks should be added to the free list.
released	Decrements the count of memory blocks on the free list.
saved	Increments the count of memory blocks on the free list.

Requirements

Header: <allocators>

Namespace: stdext

max_variable_size::allocated

Increments the count of allocated memory blocks.

```
C++
```

```
void allocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

This member function adds *_Nx* to the stored value `_Nallocs`. This member function is called after each successful call by `cache_freelist::allocate` to operator **`new`**. The argument *_Nx* is the number of memory blocks in the chunk allocated by operator **`new`**.

max_variable_size::deallocated

Decrements the count of allocated memory blocks.

```
C++
```

```
void deallocated(std::size_t _Nx = 1);
```

Parameters

_Nx
The increment value.

Remarks

The member function subtracts *_Nx* from the stored value `_Nallocs`. This member function is called after each call by `cache_freelist::deallocate` to operator **`delete`**. The argument *_Nx* is the number of memory blocks in the chunk deallocated by operator **`delete`**.

max_variable_size::full

Returns a value that specifies whether more memory blocks should be added to the free list.

```
C++

bool full();
```

Return Value

`true` if `_Nallocs / 16 + 16 <= _Nblocks`.

Remarks

This member function is called by `cache_freelist::deallocate`. If the call returns `true`, `deallocate` puts the memory block on the free list; if it returns false, `deallocate` calls operator `delete` to deallocate the block.

max_variable_size::max_variable_size

Constructs an object of type `max_variable_size`.

```
C++

max_variable_size();
```

Remarks

The constructor initializes the stored values `_Nblocks` and `_Nallocs` to zero.

max_variable_size::released

Decrements the count of memory blocks on the free list.

```
C++

void released();
```

Remarks

This member function decrements the stored value `_Nblocks`. The `released` member function of the current max class is called by `cache_freelist::allocate` whenever it removes a memory block from the free list.

max_variable_size::saved

Increments the count of memory blocks on the free list.

```
C++
```

```
void saved();
```

Remarks

This member function increments the stored value `_Nblocks`. This member function is called by `cache_freelist::deallocate` whenever it puts a memory block on the free list.

See also

<allocators>

rts_alloc Class

Article • 08/03/2021

The rts_alloc class template describes a filter that holds an array of cache instances and determines which instance to use for allocation and deallocation at runtime instead of at compile time.

Syntax

C++

```
template <class Cache>
class rts_alloc
```

Parameters

Cache
The type of cache instances contained in the array. It can be cache_chunklist, cache_freelist, or cache_suballoc.

Remarks

This class template holds multiple block allocator instances and determines which instance to use for allocation or deallocation at runtime instead of at compile time. It is used with compilers that cannot compile rebind.

Member functions

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

rts_alloc::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

The member function returns `caches[_IDX].allocate(count)`, where the index `_IDX` is determined by the requested block size *count*, or, if *count* is too large, it returns `operator new(count)`. `cache`, which represents the cache object.

rts_alloc::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

The member function calls `caches[_IDX].deallocate(ptr, count)`, where the index `_IDX` is determined by the requested block size *count*, or, if *count* is too large, it returns `operator delete(ptr)`.

rts_alloc::equals

Compares two caches for equality.

```
C++

bool equals(const sync<_Cache>& _Other) const;
```

Parameters

_Cache
The cache object associated with the filter.

_Other
The cache object to compare for equality.

Remarks

`true` if the result of `caches[0].equals(other.caches[0])`; otherwise, **`false`**. `caches` represents the array of cache objects.

See also

ALLOCATOR_DECL
<allocators>

sync_none Class

Article • 08/03/2021

Describes a synchronization filter that provides no synchronization.

Syntax

```
C++
```

```
template <class Cache>
class sync_none
```

Parameters

`Cache`

The type of cache associated with the synchronization filter. It can be cache_chunklist, cache_freelist, or cache_suballoc.

Member functions

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_none::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Remarks

The member function returns `cache.allocate(count)`, where `cache` is the cache object.

sync_none::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

The member function calls `cache.deallocate(ptr, count)`, where `cache` represents the cache object.

sync_none::equals

Compares two caches for equality.

```
C++
```

```
bool equals(const sync<Cache>& Other) const;
```

Parameters

Cache
The cache object of the synchronization filter.

Other
The cache object to compare for equality.

Return Value

The member function always returns `true`.

Remarks

See also

<allocators>

sync_per_container Class

Article • 08/03/2021

Describes a synchronization filter that provides a separate cache object for each allocator object.

Syntax

```
C++
```

```
template <class Cache>
class sync_per_container
    : public Cache
```

Parameters

Cache
The type of cache associated with the synchronization filter. It can be cache_chunklist, cache_freelist, or cache_suballoc.

Member functions

Member function	Description
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_per_container::equals

Compares two caches for equality.

```
C++
```

```
bool equals(const sync_per_container<Cache>& Other) const;
```

Parameters

Cache
The cache object of the synchronization filter.

Other
The cache object to compare for equality.

Return Value

The member function always returns `false`.

Remarks

See also

<allocators>

sync_per_thread Class

Article • 08/03/2021

Describes a synchronization filter that provides a separate cache object for each thread.

Syntax

```
C++
```

```
template <class Cache>
class sync_per_thread
```

Parameters

Cache
The type of cache associated with the synchronization filter. It can be cache_chunklist, cache_freelist, or cache_suballoc.

Remarks

Allocators that use `sync_per_thread` can compare equal even though blocks allocated in one thread cannot be deallocated from another thread. When using one of these allocators memory blocks allocated in one thread should not be made visible to other threads. In practice this means that a container that uses one of these allocators should only be accessed by a single thread.

Member functions

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_per_thread::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Remarks

The member function returns the result of a call to `cache::allocate(count)` on the cache object belonging to the current thread. If no cache object has been allocated for the current thread, it first allocates one.

sync_per_thread::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

The member function calls `deallocate` on the cache object belonging to the current thread. If no cache object has been allocated for the current thread, it first allocates one.

sync_per_thread::equals

Compares two caches for equality.

```
C++
```

```
bool equals(const sync<Cache>& Other) const;
```

Parameters

Cache
The cache object of the synchronization filter.

Other
The cache object to compare for equality.

Return Value

`false` if no cache object has been allocated for this object or for *Other* in the current thread. Otherwise it returns the result of applying `operator==` to the two cache objects.

Remarks

See also

<allocators>

sync_shared Class

Article • 08/03/2021

Describes a synchronization filter that uses a mutex to control access to a cache object that is shared by all allocators.

Syntax

```
C++
```

```
template <class Cache>
class sync_shared
```

Parameters

Cache
The type of cache associated with the synchronization filter. It can be cache_chunklist, cache_freelist, or cache_suballoc.

Member functions

Member function	Description
allocate	Allocates a block of memory.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
equals	Compares two caches for equality.

Requirements

Header: <allocators>

Namespace: stdext

sync_shared::allocate

Allocates a block of memory.

```
C++
```

```
void *allocate(std::size_t count);
```

Parameters

count
The number of elements in the array to be allocated.

Return Value

A pointer to the allocated object.

Remarks

The member function locks the mutex, calls `cache.allocate(count)`, unlocks the mutex, and returns the result of the earlier call to `cache.allocate(count)`. `cache` represents the current cache object.

sync_shared::deallocate

Frees a specified number of objects from storage beginning at a specified position.

```
C++
```

```
void deallocate(void* ptr, std::size_t count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

This member function locks the mutex, calls `cache.deallocate(ptr, count)`, where `cache` represents the cache object, and then unlocks the mutex.

sync_shared::equals

Compares two caches for equality.

```
C++
```

```
bool equals(const sync_shared<Cache>& Other) const;
```

Parameters

Cache
The type of cache associated with the synchronization filter.

Other
The cache to compare for equality.

Return Value

`true` if the result of `cache.equals(Other.cache)`, where `cache` represents the cache object, is `true`; otherwise, `false`.

Remarks

See also

<allocators>

`<any>`;

Article • 12/06/2021

Defines the class `std::any` and several supporting functions and classes.

Requirements

Header: `<any>`

Namespace: `std`

Standard: C++17 (Use at least `/std:c++17` to compile.)

Members

Functions

Name	Description
any_cast	Makes an object into an `any`.
make_any	Takes values and creates an `any` object.
swap	Exchanges the elements of two `any` objects.

Classes

Name	Description
any	An `any` instance either stores a type that satisfies the constructor requirements or has no value.
bad_any_cast	Objects thrown by a failed `any_cast`.

See also

Header files reference

<any> functions

Article • 12/06/2021

The <any> header declares several free functions for working with the any class.

Functions

any_cast	Makes an object into an `any`.
make_any	Takes values and creates an `any` object.
swap	Exchanges the elements of two `any` objects.

any_cast

Makes an object into an `any`.

C++

```
template<class T>
    T any_cast(const any& operand);
template<class T>
    T any_cast(any& operand);
template<class T>
    T any_cast(any&& operand);
template<class T>
    const T* any_cast(const any* operand) noexcept;
template<class T>
    T* any_cast(any* operand) noexcept;
```

make_any

Takes values and creates an `any` object.

C++

```
template <class T, class... Args>
    any make_any(Args&& ...args);
template <class T, class U, class... Args>
    any make_any(initializer_list<U> il, Args&& ...args);
```

`swap`

Exchanges the elements of two `any` objects.

```
C++
```

```
void swap(any& left, any& right) noexcept;
```

Parameters

`left`
An object of type `any`.

`right`
An object of type `any`.

Requirements

Header: <any>

Namespace: `std`

Standard: C++17 (Use at least `/std:c++17` to compile.)

See also

<any>
any class
bad_any_cast

`any` class

Article • 09/21/2021

An `any` object either stores an instance of a type that satisfies the constructor requirements, or it has no value. Whether it has a stored instance, or no value, is called the *state* of the `any` object.

The stored instance is called the *contained value*. Two `any` objects have the same state if either they both have no value, or both have a contained value and those values are the same.

Syntax

```
C++

class any;
```

Members

Constructors

Name	Description
any	Constructs an object of type `any`.

Functions

Name	Description
emplace	Sets an `any` value.
has_value	Returns `true` if `any` has a value.
reset	Resets an `any`.
swap	Exchanges two `any` objects.
type	Returns the `any` type.

Operators

Name	Description
operator=	Replaces the `any` with a copy of another `any`.

`any`

Constructs an object of type `any`. Also includes a destructor.

C++

```
constexpr any() noexcept;
any(const any& other);
any(any&& other) noexcept;
template <class T>
    any(T&& value);
template <class T, class... Args>
    explicit any(in_place_type_t<T>, Args&&...);
template <class T, class U, class... Args>
    explicit any(in_place_type_t<T>, initializer_list<U>, Args&&...);

~any();
```

`emplace`

Sets an `any` value.

C++

```
template <class T, class... Args>
    decay_t<T>& emplace(Args&& ...);
template <class T, class U, class... Args>
    decay_t<T>& emplace(initializer_list<U>, Args&&...);
```

`has_value`

Returns **`true`** if the `any` object has a value.

C++

```
bool has_value() const noexcept;
```

`operator=`

Replaces the `any` content with a copy of another `any`.

```
C++
```

```
any& operator=(const any& right);
any& operator=(any&& right) noexcept;
template <class T>
    any& operator=(T&& right);
```

Parameters

`right`

The `any` being copied into this `any`.

`reset`

Resets an `any`.

```
C++
```

```
void reset() noexcept;
```

`swap`

Exchanges two `any` objects.

```
C++
```

```
void swap(any& rhs) noexcept;
```

`type`

Returns the `any` type.

```
C++
```

```
const type_info& type() const noexcept;
```

Requirements

Header: <any>

Namespace: `std`

Standard: C++17 (Use at least `/std:c++17` to compile.)

See also

<any>
any_cast
make_any
swap
bad_any_cast

<array>

Article • 06/15/2022

Defines the container class template **array** and several supporting templates.

Requirements

Header: <array>

Namespace: std

> ⓘ Note
>
> The <array> library also uses the `#include <initializer_list>` statement.

Members

Classes

Name	Description
array	Stores a fixed-length sequence of elements.
tuple_element	Wraps the type of an array element.
tuple_size	Wraps the size of an array element.

Operators

Name	Description
operator==	array comparison, equal
operator!=	array comparison, not equal
operator<	array comparison, less than
operator>=	array comparison, greater than or equal
operator>	array comparison, greater than
operator<=	array comparison, less than or equal

Functions

Name	Description
get	Get specified array element.
swap	Exchanges the contents of one array with the contents of another array.

See also

<tuple>
Header Files Reference

`<array>` functions

Article • 12/07/2021

The <array> header includes two non-member functions, `get` and `swap`, that operate on **array** objects.

get
swap

get

Returns a reference to the specified element of the array.

C++

```
template <int Index, class T, size_t N>
constexpr T& get(array<T, N>& arr) noexcept;

template <int Index, class T, size_t N>
constexpr const T& get(const array<T, N>& arr) noexcept;

template <int Index, class T, size_t N>
constexpr T&& get(array<T, N>&& arr) noexcept;
```

Parameters

Index
The element offset.

T
The type of an element.

N
The number of elements in the array.

arr
The array to select from.

Example

C++

```
#include <array>
#include <iostream>

using namespace std;

typedef array<int, 4> MyArray;

int main()
{
    MyArray c0 { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& e : c0)
    {
        cout << " " << e;
    }
    cout << endl;

    // display odd elements " 1 3"
    cout << " " << get<1>(c0);
    cout << " " << get<3>(c0) << endl;
}
```

Output

```
0 1 2 3
1 3
```

swap

A non-member template specialization of `std::swap` that swaps two **array** objects.

C++

```
template <class Ty, std::size_t N>
void swap(array<Ty, N>& left, array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left

The first array to swap.

right

The second array to swap.

Remarks

The template function executes `left.swap(right)`.

Example

C++

```
// std__array__swap.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = { 4, 5, 6, 7 };
    c0.swap(c1);

    // display swapped contents " 4 5 6 7"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    swap(c0, c1);

    // display swapped contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4 5 6 7
0 1 2 3
```

See also

<array>

`<array>` operators

Article • 12/07/2021

The <array> header includes these **array** non-member comparison template functions.

operator!=
operator>
operator>=
operator<
operator<=
operator==

operator!=

Array comparison, not equal.

C++

```
template <Ty, std::size_t N>
bool operator!=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left
Left container to compare.

right
Right container to compare.

Remarks

The template function returns `!(left == right)`.

Example

C++

```
// std__array__operator_ne.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 != c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 != c1);
    std::cout << std::endl;

    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
false
true
```

`operator<`

Array comparison, less than.

C++

```
template <Ty, std::size_t N>
bool operator<(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left
Left container to compare.

right
Right container to compare.

Remarks

The template function overloads `operator<` to compare two objects of class template array Class. The function returns `lexicographical_compare(left.begin(), left.end(), right.begin())`.

Example

C++

```
// std__array__operator_lt.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};
```

```
// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 < c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 < c1);
    std::cout << std::endl;

    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
false
true
```

`operator<=`

Array comparison, less than or equal.

C++

```
template <Ty, std::size_t N>
bool operator<=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty

The type of an element.

N

The size of the array.

left

Left container to compare.

right
Right container to compare.

Remarks

The template function returns `!(right < left)`.

Example

C++

```
// std__array__operator_le.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 <= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 <= c0);
    std::cout << std::endl;

    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
```

```
true
false
```

operator==

Array comparison, equal.

```
C++
```

```
template <Ty, std::size_t N>
bool operator==(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left
Left container to compare.

right
Right container to compare.

Remarks

The template function overloads `operator==` to compare two objects of class template array Class. The function returns `equal(left.begin(), left.end(), right.begin())`.

Example

```
C++
```

```
// std__array__operator_eq.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
```

```
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 == c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 == c1);
    std::cout << std::endl;

    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
true
false
```

operator>

Array comparison, greater than.

C++

```
template <Ty, std::size_t N>
bool operator>(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left
Left container to compare.

right
Right container to compare.

Remarks

The template function returns `(right < left)`.

Example

C++

```
// std__array__operator_gt.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 > c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 > c0);
    std::cout << std::endl;
```

```
    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
false
true
```

`operator>=`

Array comparison, greater than or equal.

C++

```
template <Ty, std::size_t N>
bool operator>=(
    const array<Ty, N>& left,
    const array<Ty, N>& right);
```

Parameters

Ty
The type of an element.

N
The size of the array.

left
Left container to compare.

right
Right container to compare.

Remarks

The template function returns `!(left < right)`.

Example

C++

```
// std__array__operator_ge.cpp
// compile with: /EHsc
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
    {
    Myarray c0 = {0, 1, 2, 3};

// display contents " 0 1 2 3"
    for (Myarray::const_iterator it = c0.begin();
        it != c0.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

    Myarray c1 = {4, 5, 6, 7};

// display contents " 4 5 6 7"
    for (Myarray::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " " << *it;
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 >= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 >= c1);
    std::cout << std::endl;

    return (0);
    }
```

Output

```
0 1 2 3
4 5 6 7
true
false
```

See also

<array>

`array` Class (C++ Standard Library)

Article • 02/07/2023

Describes an object that controls a sequence of length `N` of elements of type `Ty`. The sequence is stored as an array of `Ty`, contained in the `array<Ty, N>` object.

Syntax

```
C++
```

```
template <class Ty, std::size_t N>
class array;
```

Parameters

`Ty`
The type of an element.

`N`
The number of elements.

Members

Type Definition	Description
const_iterator	The type of a constant iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.
const_reverse_iterator	The type of a constant reverse iterator for the controlled sequence.
difference_type	The type of a signed distance between two elements.
iterator	The type of an iterator for the controlled sequence.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
reverse_iterator	The type of a reverse iterator for the controlled sequence.

Type Definition	Description
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.

Member Function	Description
array	Constructs an array object.
assign	(Obsolete. Use `fill`.) Replaces all elements.
at	Accesses an element at a specified position.
back	Accesses the last element.
begin	Designates the beginning of the controlled sequence.
cbegin	Returns a random-access const iterator to the first element in the array.
cend	Returns a random-access const iterator that points just beyond the end of the array.
crbegin	Returns a const iterator to the first element in a reversed array.
crend	Returns a const iterator to the end of a reversed array.
data	Gets the address of the first element.
empty	Tests whether elements are present.
end	Designates the end of the controlled sequence.
fill	Replaces all elements with a specified value.
front	Accesses the first element.
max_size	Counts the number of elements.
rbegin	Designates the beginning of the reversed controlled sequence.
rend	Designates the end of the reversed controlled sequence.
size	Counts the number of elements.
swap	Swaps the contents of two containers.

Operator	Description
array::operator=	Replaces the controlled sequence.

Operator	Description
array::operator[]	Accesses an element at a specified position.

Remarks

The type has a default constructor `array()` and a default assignment operator `operator=`, and satisfies the requirements for an `aggregate`. Therefore, objects of type `array<Ty, N>` can be initialized by using an aggregate initializer. For example,

```
C++
```

```
array<int, 4> ai = { 1, 2, 3 };
```

creates the object `ai` that holds four integer values, initializes the first three elements to the values 1, 2, and 3, respectively, and initializes the fourth element to 0.

Requirements

Header: `<array>`

Namespace: `std`

`array::array`

Constructs an array object.

```
C++
```

```
array();

array(const array& right);
```

Parameters

`right`
Object or range to insert.

Remarks

The default constructor `array()` leaves the controlled sequence uninitialized (or default initialized). You use it to specify an uninitialized controlled sequence.

The copy constructor `array(const array& right)` initializes the controlled sequence with the sequence [*right*`.begin()`, *right*`.end()`). You use it to specify an initial controlled sequence that is a copy of the sequence controlled by the array object *`right`*.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    Myarray c1(c0);

    // display contents " 0 1 2 3"
    for (const auto& it : c1)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
```

`array::assign`

Obsolete in C++11, replaced by fill. Replaces all elements.

`array::at`

Accesses an element at a specified position.

C++

```
reference at(size_type off);

constexpr const_reference at(size_type off) const;
```

Parameters

`off`
Position of element to access.

Remarks

The member functions return a reference to the element of the controlled sequence at position *`off`*. If that position is invalid, the function throws an object of class `out_of_range`.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display odd elements " 1 3"
    std::cout << " " << c0.at(1);
    std::cout << " " << c0.at(3);
    std::cout << std::endl;
```

```
    return (0);
}
```

`array::back`

Accesses the last element.

C++

```
reference back();

constexpr const_reference back() const;
```

Remarks

The member functions return a reference to the last element of the controlled sequence, which must be non-empty.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display last element " 3"
    std::cout << " " << c0.back();
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
3
```

`array::begin`

Designates the beginning of the controlled sequence.

C++

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Remarks

The member functions return a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::iterator it2 = c0.begin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::cbegin`

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const noexcept;
```

Return Value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`array::cend`

Returns a `const` iterator that addresses the location just beyond the last element in a range.

C++

```
const_iterator cend() const noexcept;
```

Return Value

A random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

C++

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`array::const_iterator`

The type of a constant iterator for the controlled sequence.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

The type describes an object that can serve as a constant random-access iterator for the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> MyArray;

int main()
{
    MyArray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    std::cout << "it1:";
    for (MyArray::const_iterator it1 = c0.begin();
        it1 != c0.end();
        ++it1) {
        std::cout << " " << *it1;
    }
    std::cout << std::endl;

    // display first element " 0"
    MyArray::const_iterator it2 = c0.begin();
    std::cout << "it2:";
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
it1: 0 1 2 3
it2: 0
```

`array::const_pointer`

The type of a constant pointer to an element.

C++

```
typedef const Ty *const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to elements of the sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_pointer ptr = &*c0.begin();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::const_reference`

The type of a constant reference to an element.

C++

```
typedef const Ty& const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_reference ref = *c0.begin();
    std::cout << " " << ref;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::const_reverse_iterator`

The type of a constant reverse iterator for the controlled sequence.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display last element " 3"
    Myarray::const_reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
3
```

`array::crbegin`

Returns a const iterator to the first element in a reversed array.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse random-access iterator addressing the first element in a reversed array or addressing what had been the last element in the unreversed array.

Remarks

With the return value of `crbegin`, the array object can't be modified.

Example

C++

```
#include <array>
#include <iostream>

int main( )
{
   using namespace std;
   array<int, 2> v1 = {1, 2};
   array<int, 2>::iterator v1_Iter;
   array<int, 2>::const_reverse_iterator v1_rIter;

   v1_Iter = v1.begin( );
   cout << "The first element of array is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.crbegin( );
   cout << "The first element of the reversed array is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of array is 1.
The first element of the reversed array is 2.
```

`array::crend`

Returns a const iterator that addresses the location succeeding the last element in a reversed array.

C++

```
const_reverse_iterator crend() const noexcept;
```

Return Value

A const reverse random-access iterator that addresses the location succeeding the last element in a reversed array (the location that had preceded the first element in the unreversed array).

Remarks

`crend` is used with a reversed array just as array::cend is used with an array.

With the return value of `crend` (suitably decremented), the array object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its array.

The value returned by `crend` shouldn't be dereferenced.

Example

C++

```
#include <array>
#include <iostream>

int main( )
{
   using namespace std;
   array<int, 2> v1 = {1, 2};
   array<int, 2>::const_reverse_iterator v1_rIter;

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

`array::data`

Gets the address of the first element.

C++

```
Ty *data();
```

```
const Ty *data() const;
```

Remarks

The member functions return the address of the first element in the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::pointer ptr = c0.data();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::difference_type`

The type of a signed distance between two elements.

C++

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It's a synonym for the type `std::ptrdiff_t`.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display distance first-last " -4"
    Myarray::difference_type diff = c0.begin() - c0.end();
    std::cout << " " << diff;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
-4
```

`array::empty`

Tests whether no elements are present.

C++

```
constexpr bool empty() const;
```

Remarks

The member function returns true only if `N == 0`.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display whether c0 is empty " false"
    std::cout << std::boolalpha << " " << c0.empty();
    std::cout << std::endl;

    std::array<int, 0> c1;

    // display whether c1 is empty " true"
    std::cout << std::boolalpha << " " << c1.empty();
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
false
true
```

array::end

Designates the end of the controlled sequence.

C++

```
reference end();

const_reference end() const;
```

Remarks

The member functions return a random-access iterator that points just beyond the end of the sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display last element " 3"
    Myarray::iterator it2 = c0.end();
    std::cout << " " << *--it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
3
```

`array::fill`

Erases an array and copies the specified elements to the empty array.

```
C++
```

```
void fill(const Type& val);
```

Parameters

`val`

The value of the element being inserted into the array.

Remarks

`fill` replaces each element of the array with the specified value.

Example

```
C++
```

```
#include <array>
#include <iostream>

int main()
{
    using namespace std;
    array<int, 2> v1 = { 1, 2 };

    cout << "v1 = ";
    for (const auto& it : v1)
    {
        std::cout << " " << it;
    }
    cout << endl;

    v1.fill(3);
    cout << "v1 = ";
    for (const auto& it : v1)
    {
        std::cout << " " << it;
    }
    cout << endl;
}
```

`array::front`

Accesses the first element.

C++

```
reference front();

constexpr const_reference front() const;
```

Remarks

The member functions return a reference to the first element of the controlled sequence, which must be non-empty.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    std::cout << " " << c0.front();
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::iterator`

The type of an iterator for the controlled sequence.

C++

```
typedef implementation-defined iterator;
```

Remarks

The type describes an object that can serve as a random-access iterator for the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> MyArray;

int main()
{
    MyArray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    std::cout << "it1:";
    for (MyArray::iterator it1 = c0.begin();
        it1 != c0.end();
        ++it1) {
        std::cout << " " << *it1;
    }
    std::cout << std::endl;

    // display first element " 0"
    MyArray::iterator it2 = c0.begin();
    std::cout << "it2:";
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
it1: 0 1 2 3

it2: 0
```

`array::max_size`

Counts the number of elements.

C++

```
constexpr size_type max_size() const;
```

Remarks

The member function returns `N`.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display (maximum) size " 4"
    std::cout << " " << c0.max_size();
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4
```

`array::operator[]`

Accesses an element at a specified position.

C++

```
reference operator[](size_type off);

constexpr const_reference operator[](size_type off) const;
```

Parameters

`off`
Position of element to access.

Remarks

The member functions return a reference to the element of the controlled sequence at position *`off`*. If that position is invalid, the behavior is undefined.

There's also a non-member get function available to get a reference to an element of an **`array`**.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
```

```
    std::cout << std::endl;

    // display odd elements " 1 3"
    std::cout << " " << c0[1];
    std::cout << " " << c0[3];
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
1 3
```

`array::operator=`

Replaces the controlled sequence.

C++

```
array<Value> operator=(array<Value> right);
```

Parameters

`right`

Container to copy.

Remarks

The member operator assigns each element of *`right`* to the corresponding element of the controlled sequence, then returns **`*this`**. You use it to replace the controlled sequence with a copy of the controlled sequence in *`right`*.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
```

```
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    Myarray c1;
    c1 = c0;

    // display copied contents " 0 1 2 3"
        // display contents " 0 1 2 3"
    for (auto it : c1)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
```

`array::pointer`

The type of a pointer to an element.

C++

```
typedef Ty *pointer;
```

Remarks

The type describes an object that can serve as a pointer to elements of the sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::pointer ptr = &*c0.begin();
    std::cout << " " << *ptr;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

array::rbegin

Designates the beginning of the reversed controlled sequence.

C++

```
reverse_iterator rbegin()noexcept;
const_reverse_iterator rbegin() const noexcept;
```

Remarks

The member functions return a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display last element " 3"
    Myarray::const_reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
3
```

`array::reference`

The type of a reference to an element.

C++

```
typedef Ty& reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::reference ref = *c0.begin();
    std::cout << " " << ref;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

array::rend

Designates the end of the reversed controlled sequence.

C++

```
reverse_iterator rend()noexcept;
const_reverse_iterator rend() const noexcept;
```

Remarks

The member functions return a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence)). Hence, it designates the end of the reverse sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display first element " 0"
    Myarray::const_reverse_iterator it2 = c0.rend();
    std::cout << " " << *--it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0
```

`array::reverse_iterator`

The type of a reverse iterator for the controlled sequence.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

The type describes an object that can serve as a reverse iterator for the controlled sequence.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display last element " 3"
    Myarray::reverse_iterator it2 = c0.rbegin();
    std::cout << " " << *it2;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
3
```

`array::size`

Counts the number of elements.

C++

```
constexpr size_type size() const;
```

Remarks

The member function returns `N`.

Example

C++

```cpp
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display size " 4"
    std::cout << " " << c0.size();
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4
```

`array::size_type`

The type of an unsigned distance between two elements.

C++

```cpp
typedef std::size_t size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It's a synonym for the type `std::size_t`.

Example

C++

```cpp
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display distance last-first " 4"
    Myarray::size_type diff = c0.end() - c0.begin();
    std::cout << " " << diff;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4
```

`array::swap`

Swaps the contents of this array with another array.

C++

```cpp
void swap(array& right);
```

Parameters

`right`

Array to swap contents with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. It performs element assignments and constructor calls proportional to `N`.

There's also a non-member swap function available to swap two `array` instances.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    Myarray c1 = { 4, 5, 6, 7 };
    c0.swap(c1);

    // display swapped contents " 4 5 6 7"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    swap(c0, c1);

    // display swapped contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4 5 6 7
```

```
0 1 2 3
```

`array::value_type`

The type of an element.

C++

```
typedef Ty value_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

C++

```
#include <array>
#include <iostream>

typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;

    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        Myarray::value_type val = it;
        std::cout << " " << val;
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
```

See also

<array>

`<atomic>`

Article • 02/01/2022

Defines classes and class templates to use to create types that support atomic operations.

Syntax

```
C++

#include <atomic>
```

Remarks

> **Note**
>
> In code that's compiled by using **/clr:pure**, this header is blocked. Both `/clr:pure` and `/clr:safe` are deprecated in Visual Studio 2017 and later versions.

An atomic operation has two key properties that help you use multiple threads to correctly manipulate an object without using `mutex` locks.

- Because an atomic operation is indivisible, a second atomic operation on the same object from a different thread can obtain the object's state only before or after the first atomic operation.
- Based on its memory_order argument, an atomic operation establishes ordering requirements for the visibility of the effects of other atomic operations in the same thread. Consequently, it inhibits compiler optimizations that violate the ordering requirements.

On some platforms, it might not be possible to efficiently implement atomic operations for some types without using `mutex` locks. An atomic type is *`lock-free`* if no atomic operations on that type use locks.

`C++11`: In signal-handlers, you can perform atomic operations on an object `obj` if `obj.is_lock_free()` or `atomic_is_lock_free(x)` are `true`.

The class atomic_flag provides a minimal atomic type that holds a `bool` flag. Its operations are always lock-free.

The class template `atomic<T>` stores an object of its argument type `T` and provides atomic access to that stored value. You can instantiate it by using any type that can be copied by using memcpy and tested for equality by using memcmp. In particular, you can use it with user-defined types that meet these requirements and, in many cases, with floating-point types.

The template also has a set of specializations for integral types and a partial specialization for pointers. These specializations provide additional operations that aren't available through the primary template.

Pointer Specializations

The `atomic<T *>` partial specializations apply to all pointer types. They provide methods for pointer arithmetic.

Integral Specializations

The `atomic<integral>` specializations apply to all integral types. They provide additional operations that aren't available through the primary template.

Each `atomic<integral>` type has a corresponding macro that you can use in an `if directive` to determine at compile time whether operations on that type are lock-free. If the value of the macro is zero, operations on the type aren't lock-free. If the value is 1, operations might be lock-free, and a runtime check is required. If the value is 2, operations are lock-free. You can use the function `atomic_is_lock_free` to determine at runtime whether operations on the type are lock-free.

For each of the integral types, there's a corresponding named atomic type that manages an object of that integral type. Each `atomic_integral` type has the same set of member functions as the corresponding instantiation of `atomic<T>` and can be passed to any of the non-member atomic functions.

`atomic_integral` Type	Integral Type	`atomic_is_lock_free` Macro
`atomic_char`	`char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_schar`	`signed char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_uchar`	`unsigned char`	`ATOMIC_CHAR_LOCK_FREE`

`atomic_integral` Type	Integral Type	`atomic_is_lock_free` Macro
`atomic_char16_t`	**`char16_t`**	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	**`char32_t`**	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	**`wchar_t`**	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	**`short`**	`ATOMIC_SHORT_LOCK_FREE`
`atomic_ushort`	**`unsigned short`**	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	**`int`**	`ATOMIC_INT_LOCK_FREE`
`atomic_uint`	**`unsigned int`**	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	**`long`**	`ATOMIC_LONG_LOCK_FREE`
`atomic_ulong`	**`unsigned long`**	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	**`long long`**	`ATOMIC_LLONG_LOCK_FREE`
`atomic_ullong`	**`unsigned long long`**	`ATOMIC_LLONG_LOCK_FREE`

`Typedef` names exist for specializations of the atomic template for some of the types that are defined in the header `<inttypes.h>`.

Atomic Type	`Typedef` Name
`atomic_int8_t`	`atomic<int8_t>`
`atomic_uint8_t`	`atomic<uint8_t>`
`atomic_int16_t`	`atomic<int16_t>`
`atomic_uint16_t`	`atomic<uint16_t>`
`atomic_int32_t`	`atomic<int32_t>`
`atomic_uint32_t`	`atomic<uint32_t>`
`atomic_int64_t`	`atomic<int64_t>`
`atomic_uint64_t`	`atomic<uint64_t>`
`atomic_int_least8_t`	`atomic<int_least8_t>`
`atomic_uint_least8_t`	`atomic<uint_least8_t>`
`atomic_int_least16_t`	`atomic<int_least16_t>`

Atomic Type	`Typedef` Name
`atomic_uint_least16_t`	`atomic<uint_least16_t>`
`atomic_int_least32_t`	`atomic<int_least32_t>`
`atomic_uint_least32_t`	`atomic<uint_least32_t>`
`atomic_int_least64_t`	`atomic<int_least64_t>`
`atomic_uint_least64_t`	`atomic<uint_least64_t>`
`atomic_int_fast8_t`	`atomic<int_fast8_t>`
`atomic_uint_fast8_t`	`atomic<uint_fast8_t>`
`atomic_int_fast16_t`	`atomic<int_fast16_t>`
`atomic_uint_fast16_`	`atomic<uint_fast16_t>`
`atomic_int_fast32_t`	`atomic<int_fast32_t>`
`atomic_uint_fast32_t`	`atomic<uint_fast32_t>`
`atomic_int_fast64_t`	`atomic<int_fast64_t>`
`atomic_uint_fast64_t`	`atomic<uint_fast64_t>`
`atomic_intptr_t`	`atomic<intptr_t>`
`atomic_uintptr_t`	`atomic<uintptr_t>`
`atomic_size_t`	`atomic<size_t>`
`atomic_ptrdiff_t`	`atomic<ptrdiff_t>`
`atomic_intmax_t`	`atomic<intmax_t>`
`atomic_uintmax_t`	`atomic<uintmax_t>`

Structs

Name	Description
atomic Structure	Describes an object that performs `atomic` operations on a stored value.
atomic_flag Structure	Describes an object that atomically sets and clears a `bool` flag.

Enums

Name	Description
memory_order Enum	Supplies symbolic names for synchronization operations on memory locations. These operations affect how assignments in one thread become visible in another.

Functions

In the following list, the functions that don't end in `_explicit` have the semantics of the corresponding `_explicit`, except that they have the implicit memory_order arguments of `memory_order_seq_cst`.

Name	Description
atomic_compare_exchange_strong	Performs an *`atomic compare and exchange`* operation.
atomic_compare_exchange_strong_explicit	Performs an *`atomic compare and exchange`* operation.
atomic_compare_exchange_weak	Performs a *`weak atomic compare and exchange`* operation.
atomic_compare_exchange_weak_explicit	Performs a *`weak atomic compare and exchange`* operation.
atomic_exchange	Replaces a stored value.
atomic_exchange_explicit	Replaces a stored value.
atomic_fetch_add	Adds a specified value to an existing stored value.
atomic_fetch_add_explicit	Adds a specified value to an existing stored value.
atomic_fetch_and	Performs a bitwise "and" (`&`) on a specified value and an existing stored value.
atomic_fetch_and_explicit	Performs a bitwise "and" (`&`) on a specified value and an existing stored value.
atomic_fetch_or	Performs a bitwise "or" (`|`) on a specified value and an existing stored value.
atomic_fetch_or_explicit	Performs a bitwise "or" (`|`) on a specified value and an existing stored value.
atomic_fetch_sub	Subtracts a specified value from an existing stored value.

Name	Description
atomic_fetch_sub_explicit	Subtracts a specified value from an existing stored value.
atomic_fetch_xor	Performs a bitwise "exclusive or" (`^`) on a specified value and an existing stored value.
atomic_fetch_xor_explicit	Performs a bitwise "exclusive or" (`^`) on a specified value and an existing stored value.
atomic_flag_clear	Sets the flag in an `atomic_flag` object to **`false`**.
atomic_flag_clear_explicit	Sets the flag in an `atomic_flag` object to **`false`**.
atomic_flag_test_and_set	Sets the flag in an `atomic_flag` object to **`true`**.
atomic_flag_test_and_set_explicit	Sets the flag in an `atomic_flag` object to **`true`**.
atomic_init	Sets the stored value in an `atomic` object.
atomic_is_lock_free	Specifies whether atomic operations on a specified object are lock-free.
atomic_load	Atomically retrieves a value.
atomic_load_explicit	Atomically retrieves a value.
atomic_signal_fence	Acts as a *`fence`* that establishes memory ordering requirements between fences in a calling thread that has signal handlers executed in the same thread.
atomic_store	Atomically stores a value.
atomic_store_explicit	Atomically stores a value.
atomic_thread_fence	Acts as a *`fence`* that establishes memory ordering requirements with respect to other fences.
kill_dependency	Breaks a possible dependency chain.

See also

Header Files Reference

C++ Standard Library Reference

`atomic` Structure

Article • 06/09/2022

Describes an object that performs `atomic` operations on a stored value of type *`Ty`*.

Syntax

```
C++
```

```
template <class Ty>
struct atomic;
```

Members

Member	Description
Constructor	
atomic	Constructs an atomic object.
Operators	
atomic::operator Ty	Reads and returns the stored value. (atomic::load)
atomic::operator=	Uses a specified value to replace the stored value. (atomic::store)
atomic::operator++	Increments the stored value. Used only by integral and pointer specializations.
atomic::operator+=	Adds a specified value to the stored value. Used only by integral and pointer specializations.
atomic::operator--	Decrements the stored value. Used only by integral and pointer specializations.
atomic::operator-=	Subtracts a specified value from the stored value. Used only by integral and pointer specializations.
atomic::operator&=	Performs a bitwise "and" (`&`) on a specified value and the stored value. Used only by integral specializations.
atomic::operator|=	Performs a bitwise "or" (`|`) on a specified value and the stored value. Used only by integral specializations.

Member	Description
atomic::operator^=	Performs a bitwise "exclusive or" (`^`) on a specified value and the stored value. Used only by integral specializations.
Functions	
compare_exchange_strong	Performs an *`atomic_compare_and_exchange`* operation on **`this`** and returns the result.
compare_exchange_weak	Performs a *`weak_atomic_compare_and_exchange`* operation on **`this`** and returns the result.
fetch_add	Adds a specified value to the stored value.
fetch_and	Performs a bitwise "and" (`&`) on a specified value and the stored value.
fetch_or	Performs a bitwise "or" (`|`) on a specified value and the stored value.
fetch_sub	Subtracts a specified value from the stored value.
fetch_xor	Performs a bitwise "exclusive or" (`^`) on a specified value and the stored value.
is_lock_free	Specifies whether `atomic` operations on **`this`** are *lock free*. An `atomic` type is *lock free* if no `atomic` operations on that type use locks.
load	Reads and returns the stored value.
store	Uses a specified value to replace the stored value.

Remarks

The type *`Ty`* must be *trivially copyable*. That is, using memcpy to copy its bytes must produce a valid *`Ty`* object that compares equal to the original object. The compare_exchange_weak and compare_exchange_strong member functions use memcmp to determine whether two *`Ty`* values are equal. These functions won't use a *`Ty`*-defined `operator==`. The member functions of `atomic` use `memcpy` to copy values of type *`Ty`*.

A partial specialization, `atomic<Ty*>`, exists for all pointer types. The specialization enables the addition of an offset to the managed pointer value or the subtraction of an offset from it. The arithmetic operations take an argument of type `ptrdiff_t` and adjust

that argument according to the size of *Ty* to be consistent with ordinary address arithmetic.

A specialization exists for every integral type except **bool**. Each specialization provides a rich set of methods for atomic arithmetic and logical operations.

`atomic<char>`

`atomic<signed char>`

`atomic<unsigned char>`

`atomic<char16_t>`

`atomic<char32_t>`

`atomic<wchar_t>`

`atomic<short>`

`atomic<unsigned short>`

`atomic<int>`

`atomic<unsigned int>`

`atomic<long>`

`atomic<unsigned long>`

`atomic<long long>`

`atomic<unsigned long long>`

Integral specializations are derived from corresponding `atomic_integral` types. For example, `atomic<unsigned int>` is derived from `atomic_uint`.

Requirements

Header: `<atomic>`

Namespace: `std`

`atomic::atomic`

Constructs an atomic object.

C++

```
atomic();
atomic( const atomic& );
atomic( Ty Value ) noexcept;
```

Parameters

`Value`
Initialization value.

Remarks

Atomic objects can't be copied or moved.

Objects that are instantiations of `atomic<Ty>` can be initialized only by the constructor that takes an argument of type `Ty` and not by using aggregate initialization. However, `atomic_integral` objects can be initialized only by using aggregate initialization.

```
C++
```

```
atomic<int> ai0 = ATOMIC_VAR_INIT(0);
atomic<int> ai1(0);
```

`atomic::operator Ty`

The operator for the type specified to the template, `atomic<Ty>`. Retrieves the stored value in `*this`.

```
C++
```

```
atomic<Ty>::operator Ty() const volatile noexcept;
atomic<Ty>::operator Ty() const noexcept;
```

Remarks

This operator applies the `memory_order_seq_cst` memory_order.

`atomic::operator=`

Stores a specified value.

```
C++
```

```
Ty operator=(
   Ty Value
) volatile noexcept;
Ty operator=(
```

```
    Ty Value
) noexcept;
```

Parameters

Value

A *Ty* object.

Return Value

Returns *Value*.

atomic::operator++

Increments the stored value. Used only by integral and pointer specializations.

```
C++
```

```
Ty atomic<Ty>::operator++(int) volatile noexcept;
Ty atomic<Ty>::operator++(int) noexcept;
Ty atomic<Ty>::operator++() volatile noexcept;
Ty atomic<Ty>::operator++() noexcept;
```

Return Value

The first two operators return the incremented value; the last two operators return the value before the increment. The operators use the `memory_order_seq_cst` memory_order.

atomic::operator+=

Adds a specified value to the stored value. Used only by integral and pointer specializations.

```
C++
```

```
Ty atomic<Ty>::operator+=(
    Ty Value
) volatile noexcept;
Ty atomic<Ty>::operator+=(
    Ty Value
) noexcept;
```

Parameters

Value
An integral or pointer value.

Return Value

A *Ty* object that contains the result of the addition.

Remarks

This operator uses the `memory_order_seq_cst` memory_order.

atomic::operator--

Decrements the stored value. Used only by integral and pointer specializations.

```
C++

Ty atomic<Ty>::operator--(int) volatile noexcept;
Ty atomic<Ty>::operator--(int) noexcept;
Ty atomic<Ty>::operator--() volatile noexcept;
Ty atomic<Ty>::operator--() noexcept;
```

Return Value

The first two operators return the decremented value; the last two operators return the value before the decrement. The operators use the `memory_order_seq_cst` memory_order.

atomic::operator-=

Subtracts a specified value from the stored value. Used only by integral and pointer specializations.

```
C++

Ty atomic<Ty>::operator-=(
   Ty Value
) volatile noexcept;
Ty atomic<Ty>::operator-=(
```

```
    Ty Value
) noexcept;
```

Parameters

Value
An integral or pointer value.

Return Value

A *Ty* object that contains the result of the subtraction.

Remarks

This operator uses the `memory_order_seq_cst` memory_order.

`atomic::operator&=`

Performs a bitwise "and" (`&`) on a specified value and the stored value of `*this`. Used only by integral specializations.

C++

```
atomic<Ty>::operator&= (
    Ty Value
) volatile noexcept;
atomic<Ty>::operator&= (
    Ty Value
) noexcept;
```

Parameters

Value
A value of type *Ty*.

Return Value

The result of the bitwise "and" (`&`).

Remarks

This operator performs a read-modify-write operation to replace the stored value of `*this` with a bitwise "and" (`&`) of *`Value`* and the current value that is stored in `*this`, within the constraints of the `memory_order_seq_cst` memory_order.

`atomic::operator|;=`

Performs a bitwise "or" (`|`) on a specified value and the stored value of `*this`. Used only by integral specializations.

```
C++
```

```
atomic<Ty>::operator|= (
   Ty Value
) volatile noexcept;
atomic<Ty>::operator|= (
   Ty Value
) noexcept;
```

Parameters

`Value`
A value of type *`Ty`*.

Return Value

The result of the bitwise "or" (`|`).

Remarks

This operator performs a read-modify-write operation to replace the stored value of `*this` with a bitwise "or" (`|`) of *`Value`* and the current value that is stored in `*this`, within the constraints of the `memory_order_seq_cst` memory_order constraints.

`atomic::operator^=`

Performs a bitwise "exclusive or" (`^`) on a specified value and the stored value of `*this`. Used only by integral specializations.

```
C++
```

```
atomic<Ty>::operator^= (
   Ty Value
) volatile noexcept;
atomic<Ty>::operator^= (
   Ty Value
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Return Value

The result of the bitwise "exclusive or" (^).

Remarks

This operator performs a read-modify-write operation to replace the stored value of ***this** with a bitwise "exclusive or" (^) of *Value* and the current value that is stored in ***this**, within the constraints of the memory_order_seq_cst memory_order constraints.

atomic::compare_exchange_strong

Performs an atomic compare and exchange operation on ***this**.

C++

```
bool compare_exchange_strong(
   Ty& Exp,
   Ty Value,
   memory_order Order1,
   memory_order Order2
) volatile noexcept;
bool compare_exchange_strong(
   Ty& Exp,
   Ty Value,
   memory_order Order1,
   memory_order Order2
) noexcept;
bool compare_exchange_strong(
   Ty& Exp,
   Ty Value,
   memory_order Order1 = memory_order_seq_cst
```

```
) volatile noexcept;
bool compare_exchange_strong(
   Ty& Exp,
   Ty Value,
   memory_order Order1 = memory_order_seq_cst
) noexcept;
```

Parameters

Exp

A value of type *Ty*.

Value

A value of type *Ty*.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument.

Return Value

A **bool** that indicates the result of the value comparison.

Remarks

This atomic compare and exchange operation compares the value that is stored in ***this** with *Exp*. If the values are equal, the operation replaces the value that is stored in ***this** with *Value* by using a read-modify-write operation and applying the memory order constraints that are specified by *Order1*. If the values aren't equal, the operation uses the value that is stored in ***this** to replace *Exp* and applies the memory order constraints that are specified by *Order2*.

Overloads that don't have a second `memory_order` use an implicit *Order2* that is based on the value of *Order1*. If *Order1* is `memory_order_acq_rel`, *Order2* is `memory_order_acquire`. If *Order1* is `memory_order_release`, *Order2* is `memory_order_relaxed`. In all other cases, *Order2* is equal to *Order1*.

For overloads that take two `memory_order` parameters, the value of *Order2* must not be `memory_order_release` or `memory_order_acq_rel`, and must not be stronger than the

value of *Order1*.

atomic::compare_exchange_weak

Performs a weak atomic compare and exchange operation on ***this**.

```
C++
```

```
bool compare_exchange_weak(
   Ty& Exp,
   Ty Value,
   memory_order Order1,
   memory_order Order2
) volatile noexcept;
bool compare_exchange_weak(
   Ty& Exp,
   Ty Value,
   memory_order Order1,
   memory_order Order2
) noexcept;
bool compare_exchange_weak(
   Ty& Exp,
   Ty Value,
   memory_order Order1 = memory_order_seq_cst
) volatile noexcept;
bool compare_exchange_weak(
   Ty& Exp,
   Ty Value,
   memory_order Order1 = memory_order_seq_cst
) noexcept;
```

Parameters

Exp

A value of type *Ty*.

Value

A value of type *Ty*.

Order1

First `memory_order` argument.

Order2

Second `memory_order` argument.

Return Value

A `bool` that indicates the result of the value comparison.

Remarks

This atomic compare and exchange operation compares the value that is stored in `*this` with *`Exp`*. If the values are equal, the operation replaces the value that is stored in `*this` with *`Value`* by using a read-modify-write operation and applying the memory order constraints that are specified by *`Order1`*. If the values aren't equal, the operation uses the value that is stored in `*this` to replace *`Exp`* and applies the memory order constraints that are specified by *`Order2`*.

A weak atomic compare and exchange operation performs an exchange if the compared values are equal. If the values aren't equal, the operation isn't guaranteed to perform an exchange.

Overloads that don't have a second `memory_order` use an implicit *`Order2`* that is based on the value of *`Order1`*. If *`Order1`* is `memory_order_acq_rel`, *`Order2`* is `memory_order_acquire`. If *`Order1`* is `memory_order_release`, *`Order2`* is `memory_order_relaxed`. In all other cases, *`Order2`* is equal to *`Order1`*.

For overloads that take two `memory_order` parameters, the value of *`Order2`* must not be `memory_order_release` or `memory_order_acq_rel`, and must not be stronger than the value of *`Order1`*.

`atomic::exchange`

Uses a specified value to replace the stored value of `*this`.

C++

```
Ty atomic<Ty>::exchange(
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::exchange(
   Ty Value,
   memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

The stored value of `*this` before the exchange.

Remarks

This operation performs a read-modify-write operation to use *Value* to replace the value that is stored in `*this`, within the memory constraints that are specified by *Order*.

`atomic::fetch_add`

Fetches the value stored in `*this`, and then adds a specified value to the stored value.

```
C++
```

```
Ty atomic<Ty>::fetch_add (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_add (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the value stored in `*this` prior to the addition.

Remarks

The `fetch_add` method performs a read-modify-write operation to atomically add *`Value`* to the stored value in **`*this`**, and applies the memory constraints that are specified by *`Order`*.

`atomic::fetch_and`

Performs a bitwise "and" (`&`) on a value and an existing value that is stored in **`*this`**.

```
C++
```

```
Ty atomic<Ty>::fetch_and (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_and (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

`Value`
A value of type *`Ty`*.

`Order`
A `memory_order`.

Return Value

A *`Ty`* object that contains the result of the bitwise "and" (`&`).

Remarks

The `fetch_and` method performs a read-modify-write operation to replace the stored value of **`*this`** with a bitwise "and" (`&`) of *`Value`* and the current value that is stored in **`*this`**, within the memory constraints that are specified by *`Order`*.

`atomic::fetch_or`

Performs a bitwise "or" (`|`) on a value and an existing value that is stored in `*this`.

```
C++
```

```
Ty atomic<Ty>::fetch_or (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_or (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the bitwise "or" (`|`).

Remarks

The `fetch_or` method performs a read-modify-write operation to replace the stored value of `*this` with a bitwise "or" (`|`) of *Value* and the current value that is stored in `*this`, within the memory constraints that are specified by *Order*.

`atomic::fetch_sub`

Subtracts a specified value from the stored value.

```
C++
```

```
Ty atomic<Ty>::fetch_sub (
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_sub (
   Ty Value,
```

```
    memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the subtraction.

Remarks

The `fetch_sub` method performs a read-modify-write operation to atomically subtract *Value* from the stored value in `*this`, within the memory constraints that are specified by *Order*.

`atomic::fetch_xor`

Performs a bitwise "exclusive or" (`^`) on a value and an existing value that is stored in `*this`.

C++

```
Ty atomic<Ty>::fetch_xor (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) volatile noexcept;
Ty atomic<Ty>::fetch_xor (
    Ty Value,
    memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A value of type *Ty*.

Order

A `memory_order`.

Return Value

A *Ty* object that contains the result of the bitwise "exclusive or" (`^`).

Remarks

The `fetch_xor` method performs a read-modify-write operation to replace the stored value of `*this` with a bitwise "exclusive or" (`^`) of *Value* and the current value that is stored in `*this`, and applies the memory constraints that are specified by *Order*.

`atomic::is_lock_free`

Specifies whether `atomic` operations on `*this` are lock free.

```
C++
```

```
bool is_lock_free() const volatile noexcept;
```

Return Value

`true` if `atomic` operations on `*this` are lock free; otherwise, `false`.

Remarks

An `atomic` type is lock free if no `atomic` operations on that type use locks.

`atomic::load`

Retrieves the stored value in `*this`, within the specified memory constraints.

```
C++
```

```
Ty atomic::load(
   memory_order Order = memory_order_seq_cst
) const volatile noexcept;
Ty atomic::load(
   memory_order Order = memory_order_seq_cst
) const noexcept;
```

Parameters

Order

A memory_order. *Order* must not be memory_order_release or memory_order_acq_rel.

Return Value

The retrieved value that is stored in ***this**.

atomic::store

Stores a specified value.

```
C++

void atomic<Ty>::store(
   Ty Value,
   memory_order Order = memory_order_seq_cst
) volatile noexcept;
void atomic<Ty>::store(
   Ty Value,
   memory_order Order = memory_order_seq_cst
) noexcept;
```

Parameters

Value

A *Ty* object.

Order

A memory_order constraint.

Remarks

This member function atomically stores *Value* in ***this**, within the memory constraints that are specified by *Order*.

See also

<atomic>

Header Files Reference

atomic_flag Structure

Article • 06/09/2022

Describes an object that atomically sets and clears a `bool` flag. Operations on atomic flags are always lock-free.

Syntax

```
C++

struct atomic_flag;
```

Members

Public Methods

Name	Description
clear	Sets the stored flag to `false`.
test_and_set	Sets the stored flag to `true` and returns the initial flag value.

Remarks

`atomic_flag` objects can be passed to the non-member functions atomic_flag_clear, atomic_flag_clear_explicit, atomic_flag_test_and_set, and atomic_flag_test_and_set_explicit. They can be initialized by using the value `ATOMIC_FLAG_INIT`.

Requirements

Header: <atomic>

Namespace: std

atomic_flag::clear

Sets the `bool` flag that is stored in `*this` to `false`, within the specified memory_order constraints.

```
C++
```

```
void atomic_flag::clear(memory_order Order = memory_order_seq_cst) volatile
noexcept;
void atomic_flag::clear(memory_order Order = memory_order_seq_cst) noexcept;
```

Parameters

Order
A memory_order.

atomic_flag::test_and_set

Sets the `bool` flag that is stored in `*this` to `true`, within the specified memory_order constraints.

```
C++
```

```
bool atomic_flag::test_and_set(memory_order Order = memory_order_seq_cst)
volatile noexcept;
bool atomic_flag::test_and_set(memory_order Order = memory_order_seq_cst)
noexcept;
```

Parameters

Order
A memory_order.

Return Value

The initial value of the flag that is stored in `*this`.

See also

<atomic>

`<atomic>` functions

Article • 01/25/2022

atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit
atomic_exchange
atomic_exchange_explicit
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag_clear
atomic_flag_clear_explicit
atomic_flag_test_and_set
atomic_flag_test_and_set_explicit
atomic_init
atomic_is_lock_free
atomic_load
atomic_load_explicit
atomic_signal_fence
atomic_store
atomic_store_explicit
atomic_thread_fence
kill_dependency

`atomic_compare_exchange_strong`

Performs an atomic compare and exchange operation.

```
C++
```

```
template <class Ty>
inline bool atomic_compare_exchange_strong(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Value) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;
```

Parameters

Atom

A pointer to an *atomic* object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Return Value

`true` if the values are equal, otherwise **`false`**.

Remarks

This method performs an atomic compare and exchange operation by using implicit memory_order.memory_order_seq_cst arguments. For more information, see atomic_compare_exchange_strong_explicit.

atomic_compare_exchange_strong_explicit

Performs an *atomic compare and exchange* operation.

C++

```
template <class T>
inline bool atomic_compare_exchange_strong_explicit(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
```

```
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong_explicit(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Exp

A pointer to a value of type `Ty`.

Value

A value of type `Ty`.

Order1

First memory_order argument.

Order2

Second `memory_order` argument. The value of *Order2* cannot be `memory_order_release` or `memory_order_acq_rel`, it cannot be stronger than the value of *Order1*.

Return Value

`true` if the values are equal, otherwise **`false`**.

Remarks

An *atomic compare and exchange operation* compares the value that is stored in the object that is pointed to by *Atom* against the value that is pointed to by *Exp*. If the values are equal, the value that is stored in the object that is pointed to by *Atom* is replaced with *Value* by using a read-modify-write operation and applying the memory order constraints that are specified by *Order1*. If the values are not equal, the operation replaces the value that is pointed to by *Exp* with the value that is stored in the object

that is pointed to by *Atom* and applies the memory order constraints that are specified by *Order2*.

atomic_compare_exchange_weak

Performs a *weak atomic compare and exchange* operation.

C++

```
template <class Ty>
inline bool atomic_compare_exchange_strong(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_strong(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value) noexcept;
```

Parameters

Atom
A pointer to an `atomic` object that stores a value of type `Ty`.

Exp
A pointer to a value of type `Ty`.

Value
A value of type `Ty`.

Return Value

`true` if the values are equal, otherwise **`false`**.

Remarks

This method performs a *weak atomic compare and exchange operation* that has implicit memory_order.memory_order_seq_cst arguments. For more information, see atomic_compare_exchange_weak_explicit.

`atomic_compare_exchange_weak_explicit`

Performs a *weak atomic compare and exchange* operation.

```
C++
```

```
template <class Ty>
inline bool atomic_compare_exchange_weak_explicit(
    volatile atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;

template <class Ty>
inline bool atomic_compare_exchange_weak_explicit(
    atomic<Ty>* Atom,
    Ty* Exp,
    Ty Value,
    memory_order Order1,
    memory_order Order2) noexcept;
```

Parameters

`Atom`

A pointer to an `atomic` object that stores a value of type `Ty`.

`Exp`

A pointer to a value of type `Ty`.

`Value`

A value of type `Ty`.

`Order1`

First memory_order argument.

`Order2`

Second `memory_order` argument. The value of *`Order2`* cannot be `memory_order_release` or `memory_order_acq_rel`, nor can it be stronger than the value of *`Order1`*.

Return Value

`true` if the values are equal, otherwise **`false`**.

Remarks

Both the strong and weak flavors of an *atomic compare and exchange operation* guarantee that they do not store the new value if the expected and current values are not equal. The strong flavor guarantees that it will store the new value if the expected and current values are equal. The weak flavor may sometimes return **false** and not store the new value even if the current and expected values are equal. In other words, the function will return **false**, but a later examination of the expected value might reveal that it did not change, and therefore should have compared as equal.

atomic_exchange

Uses *Value* to replace the stored value of *Atom*.

C++

```
template <class T>
inline Ty atomic_exchange(volatile atomic<Ty>* _Atom, Ty Value) noexcept;

template <class Ty>
inline T atomic_exchange(atomic<Ty>* Atom, Ty Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Value

A value of type `Ty`.

Return Value

The stored value of *Atom* before the exchange.

Remarks

The `atomic_exchange` function performs a read-modify-write operation to exchange the value that is stored in *Atom* with *Value*, using the memory_order.memory_order_seq_cst.

atomic_exchange_explicit

Replaces the stored value of *Atom* with *Value*.

```
C++
```

```
template <class Ty>
inline Ty atomic_exchange_explicit(
    volatile atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;

template <class Ty>
inline Ty atomic_exchange_explicit(
    atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `Ty`.

Value

A value of type `Ty`.

Order

A memory_order.

Return Value

The stored value of *Atom* before the exchange.

Remarks

The `atomic_exchange_explicit` function performs a read-modify-write operation to exchange the value that is stored in *Atom* with *Value*, within the memory constraints that are specified by *Order*.

`atomic_fetch_add`

Adds a value to an existing value that is stored in an `atomic` object.

```
C++
```

```
template <class T>
T* atomic_fetch_add(volatile atomic<T*>* Atom, ptrdiff_t Value) noexcept;
template <class T>
T* atomic_fetch_add(atomic<T*>* Atom, ptrdiff_t Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_add` function performs a read-modify-write operation to atomically add *Value* to the stored value in *Atom*, using the memory_order.memory_order_seq_cst constraint.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

C++

```
integral atomic_fetch_add(volatile atomic-integral* Atom, integral Value)
noexcept;

integral atomic_fetch_add(atomic-integral* Atom, integral Value) noexcept;
```

`atomic_fetch_add_explicit`

Adds a value to an existing value that is stored in an `atomic` object.

C++

```
template <class T>
T* atomic_fetch_add_explicit(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;

template <class T>
T* atomic_fetch_add_explicit(
    atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_add_explicit` function performs a read-modify-write operation to atomically add *Value* to the stored value in *Atom*, within the memory_order constraints that are specified by `Order`.

When the atomic type is `atomic_address`, `Value` has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
C++

integral atomic_fetch_add_explicit(
    volatile atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;

integral atomic_fetch_add_explicit(
    atomic-integral* Atom,
```

```
    integral Value,
    memory_order Order) noexcept;
```

`atomic_fetch_and`

Performs a bitwise `&` on a value and an existing value that is stored in an `atomic` object.

C++

```
template <class T>
inline T atomic_fetch_and(volatile atomic<T>* Atom, T Value) noexcept;
template <class T>
inline T atomic_fetch_and(volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

`Atom`

A pointer to an `atomic` object that stores a value of type `T`.

`Value`

A value of type `T`.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_and` function performs a read-modify-write operation to replace the stored value of *`Atom`* with a bitwise `&` of *`Value`* and the current value that is stored in *`Atom`*, using the memory_order.memory_order_seq_cst constraint.

`atomic_fetch_and_explicit`

Performs a bitwise `&` of a value and an existing value that is stored in an `atomic` object.

C++

```
template <class T>
inline T atomic_fetch_and_explicit(
```

```
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_and_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A memory_order.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_and_explicit` function performs a read-modify-write operation to replace the stored value of *Atom* with a bitwise `&` of *Value* and the current value that is stored in *Atom*, within the memory constraints that are specified by *Order*.

`atomic_fetch_or`

Performs a bitwise `or` on a value and an existing value that is stored in an `atomic` object.

C++

```
template <class T>
inline T atomic_fetch_or (volatile atomic<T>* Atom, T Value) noexcept;
template <class T>
inline T atomic_fetch_or (volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_or` function performs a read-modify-write operation to replace the stored value of *Atom* with a bitwise `or` of *Value* and the current value that is stored in *Atom*, using the memory_order.memory_order_seq_cst.

`atomic_fetch_or_explicit`

Performs a bitwise `or` on a value and an existing value that is stored in an `atomic` object.

C++

```
template <class T>
inline T atomic_fetch_or_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_or_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A memory_order.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_or_explicit` function performs a read-modify-write operation to replace the stored value of *Atom* with a bitwise `or` of *Value* and the current value that is stored in *Atom*, within the memory_order constraints specified by *Order*.

`atomic_fetch_sub`

Subtracts a value from an existing value that is stored in an `atomic` object.

C++

```
template <class T>
T* atomic_fetch_sub(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value) noexcept;

template <class T>
T* atomic_fetch_sub(
    atomic<T*>* Atom,
    ptrdiff_t Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_sub` function performs a read-modify-write operation to atomically subtract *Value* from the stored value in *Atom*, using the memory_order.memory_order_seq_cst constraint.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
C++

integral atomic_fetch_sub(volatile atomic-integral* Atom, integral Value)
noexcept;
integral atomic_fetch_sub(atomic-integral* Atom, integral Value) noexcept;
```

atomic_fetch_sub_explicit

Subtracts a value from an existing value that is stored in an `atomic` object.

```
C++

template <class T>
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>* Atom,
    ptrdiff_t Value,
    memory_order Order) noexcept;

template <class T>
T* atomic_fetch_sub_explicit(
    atomic<T*>* Atom,
    ptrdiff_t Value, memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a pointer to type `T`.

Value

A value of type `ptrdiff_t`.

Return Value

The value of the pointer contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_sub_explicit` function performs a read-modify-write operation to atomically subtract *Value* from the stored value in *Atom*, within the memory_order constraints that are specified by `Order`.

When the atomic type is `atomic_address`, *Value* has type `ptrdiff_t` and the operation treats the stored pointer as a `char *`.

This operation is also overloaded for integral types:

```
C++
```

```
integral atomic_fetch_sub_explicit(
    volatile atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;

integral atomic_fetch_sub_explicit(
    atomic-integral* Atom,
    integral Value,
    memory_order Order) noexcept;
```

`atomic_fetch_xor`

Performs a bitwise `exclusive or` on a value and an existing value that is stored in an `atomic` object.

```
C++
```

```
template <class T>
inline T atomic_fetch_xor(volatile atomic<T>* Atom, T Value) noexcept;
```

```
template <class T>
inline T atomic_fetch_xor(volatile atomic<T>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_xor` function performs a read-modify-write operation to replace the stored value of *Atom* with a bitwise `exclusive or` of *Value* and the current value that is stored in *Atom*, using the memory_order.memory_order_seq_cst.

`atomic_fetch_xor_explicit`

Performs a bitwise `exclusive or` on a value and an existing value stored in an `atomic` object.

C++

```
template <class T>
inline T atomic_fetch_xor_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;

template <class T>
inline T atomic_fetch_xor_explicit(
    volatile atomic<T>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that stores a value of type `T`.

Value

A value of type `T`.

Order

A memory_order.

Return Value

The value contained by the `atomic` object immediately before the operation was performed.

Remarks

The `atomic_fetch_xor_explicit` function performs a read-modify-write operation to replace the stored value of *Atom* with a bitwise `exclusive or` of *Value* and the current value that is stored in *Atom*, within the memory_order constraints that are specified by *Order*.

atomic_flag_clear

Sets the **bool** flag in an atomic_flag object to **false**, within the memory_order.memory_order_seq_cst.

C++

```
inline void atomic_flag_clear(volatile atomic_flag* Flag) noexcept;
inline void atomic_flag_clear(atomic_flag* Flag) noexcept;
```

Parameters

Flag

A pointer to an `atomic_flag` object.

atomic_flag_clear_explicit

Sets the `bool` flag in an atomic_flag object to `false`, within the specified memory_order constraints.

```
C++
```

```
inline void atomic_flag_clear_explicit(volatile atomic_flag* Flag,
memory_order Order) noexcept;
inline void atomic_flag_clear_explicit(atomic_flag* Flag, memory_order
Order) noexcept;
```

Parameters

Flag

A pointer to an `atomic_flag` object.

Order

A memory_order.

atomic_flag_test_and_set

Sets the `bool` flag in an atomic_flag object to `true`, within the constraints of the memory_order.memory_order_seq_cst.

```
C++
```

```
inline bool atomic_flag_test_and_set(volatile atomic_flag* Flag,) noexcept;
inline bool atomic_flag_test_and_set(atomic_flag* Flag,) noexcept;
```

Parameters

Flag

A pointer to an `atomic_flag` object.

Return Value

The initial value of *Flag*.

atomic_flag_test_and_set_explicit

Sets the `bool` flag in an atomic_flag object to `true`, within the specified memory_order constraints.

```
C++
```

```
inline bool atomic_flag_test_and_set_explicit(volatile atomic_flag* Flag,
memory_order Order) noexcept;
inline bool atomic_flag_test_and_set_explicit(atomic_flag* Flag,
memory_order Order) noexcept;
```

Parameters

`Flag`

A pointer to an `atomic_flag` object.

`Order`

A memory_order.

Return Value

The initial value of *`Flag`*.

`atomic_init`

Sets the stored value in an `atomic` object.

```
C++
```

```
template <class Ty>
inline void atomic_init(volatile atomic<Ty>* Atom, Ty Value) noexcept;
template <class Ty>
inline void atomic_init(atomic<Ty>* Atom, Ty Value) noexcept;
```

Parameters

`Atom`

A pointer to an `atomic` object that stores a value of type `Ty`.

`Value`

A value of type `Ty`.

Remarks

`atomic_init` isn't an atomic operation. It isn't thread-safe.

`atomic_is_lock_free`

Specifies whether atomic operations on an `atomic` object are *lock-free*.

C++

```
template <class T>
inline bool atomic_is_lock_free(const volatile atomic<T>* Atom) noexcept;
template <class T>
inline bool atomic_is_lock_free(const atomic<T>* Atom) noexcept;
```

Parameters

`Atom`
A pointer to an `atomic` object that stores a value of type `T`.

Return Value

`true` if atomic operations on *`Atom`* are lock-free; otherwise, **`false`**.

Remarks

An atomic type is lock-free if no atomic operations on that type use locks. If this function returns **`true`**, the type is safe to use in signal-handlers.

`atomic_load`

Retrieves the stored value in an `atomic` object.

C++

```
template <class Ty>
inline Ty atomic_load(const volatile atomic<Ty>* Atom) noexcept;
template <class Ty>
inline Ty atomic_load(const atomic<Ty>* Atom) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Return Value

The retrieved value that is stored in *Atom*.

Remarks

`atomic_load` implicitly uses the memory_order.memory_order_seq_cst.

`atomic_load_explicit`

Retrieves the stored value in an `atomic` object, within a specified memory_order.

C++

```
template <class Ty>
inline Ty atomic_load_explicit(const volatile atomic<Ty>* Atom, memory_order
Order) noexcept;
template <class Ty>
inline Ty atomic_load_explicit(const atomic<Ty>* Atom, memory_order Order)
noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Order

A memory_order. Do not use `memory_order_release` or `memory_order_acq_rel`.

Return Value

The retrieved value that is stored in *Atom*.

`atomic_signal_fence`

Acts as a *fence* — a memory synchronization primitive that enforces ordering between load/store operations — between other fences in a calling thread that have signal handlers that are executed in the same thread.

```
C++
```

```
inline void atomic_signal_fence(memory_order Order) noexcept;
```

Parameters

`Order`
A memory ordering constraint that determines fence type.

Remarks

The `Order` argument determines fence type.

Value	Description
`memory_order_relaxed`	The fence has no effect.
`memory_order_consume`	The fence is an acquire fence.
`memory_order_acquire`	The fence is an acquire fence.
`memory_order_release`	The fence is a release fence.
`memory_order_acq_rel`	The fence is both an acquire fence and a release fence.
`memory_order_seq_cst`	The fence is both an acquire fence and a release fence, and is sequentially consistent.

`atomic_store`

Atomically stores a value in an `atomic` object.

```
C++
```

```
template <class Ty>
inline Ty atomic_store_explicit(const volatile atomic<Ty>* Atom, Ty Value)
noexcept;
template <class Ty>
inline Ty atomic_store_explicit(const atomic<Ty>* Atom, T Value) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Value

A value of type `Ty`.

Remarks

`atomic_store` stores *Value* in the object that is pointed to by *Atom*, within the memory_order.memory_order_seq_cst constraint.

`atomic_store_explicit`

Atomically stores a value in an `atomic` object.

C++

```
template <class Ty>
inline Ty atomic_store_explicit(
    const volatile atomic<Ty>* Atom,
    Ty Value,
    memory_order Order) noexcept;

template <class Ty>
inline Ty atomic_store_explicit(
    const atomic<Ty>* Atom,
    T Value,
    memory_order Order) noexcept;
```

Parameters

Atom

A pointer to an `atomic` object that contains a value of type `Ty`.

Value

A value of type `Ty`.

Order

A memory_order. Do not use `memory_order_consume`, `memory_order_acquire`, or `memory_order_acq_rel`.

Remarks

`atomic_store` stores *`Value`* in the object that is pointed to by *`Atom`*, within the `memory_order` that is specified by *`Order`*.

`atomic_thread_fence`

Acts as a *fence* — a memory synchronization primitive that enforces ordering between load/store operations — without an associated atomic operation.

```
C++
```

```
inline void atomic_thread_fence(memory_order Order) noexcept;
```

Parameters

`Order`
A memory ordering constraint that determines fence type.

Remarks

The *`Order`* argument determines fence type.

Value	Description
`memory_order_relaxed`	The fence has no effect.
`memory_order_consume`	The fence is an acquire fence.
`memory_order_acquire`	The fence is an acquire fence.
`memory_order_release`	The fence is a release fence.
`memory_order_acq_rel`	The fence is both an acquire fence and a release fence.
`memory_order_seq_cst`	The fence is both an acquire fence and a release fence, and is sequentially consistent.

`kill_dependency`

Removes a dependency.

```
C++
```

```
template <class Ty>
Ty kill_dependency(Ty Arg) noexcept;
```

Parameters

Arg
A value of type `Ty`.

Return Value

The return value is *Arg*. The evaluation of *Arg* does not carry a dependency to the function call. By breaking a possible dependency chain, the function might permit the compiler to generate more efficient code.

See also

<atomic>

`<atomic>` enums

Article • 12/07/2021

memory_order Enum

Supplies symbolic names for synchronization operations on memory locations. These operations affect how assignments in one thread become visible in another.

C++

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst,
} memory_order;
```

Enumeration members

Name	Description
`memory_order_relaxed`	No ordering required.
`memory_order_consume`	A load operation acts as a consume operation on the memory location.
`memory_order_acquire`	A load operation acts as an acquire operation on the memory location.
`memory_order_release`	A store operation acts as a release operation on the memory location.
`memory_order_acq_rel`	Combines `memory_order_acquire` and `memory_order_release`.
`memory_order_seq_cst`	Combines `memory_order_acquire` and `memory_order_release`. Memory accesses that are marked as `memory_order_seq_cst` must be sequentially consistent.

See also

<atomic>

<bit>

Article • 12/07/2021

Defines functions to access, manipulate, and process individual bits and sequences of bits.

For example, there are functions to rotate bits, find the number of consecutive set or cleared bits, see if a number is an integral power of two, find the smallest number of bits to represent a number, and so on.

Requirements

Header: <bit>

Namespace: std

/std:c++20 or later is required.

Members

Types

Type	Description
endian	Specifies the endianness of scalar types.

Functions

Function	Description
bit_cast	Reinterpret the object representation from one type to another.
bit_ceil	Find the smallest power of two greater than or equal to a value.
bit_floor	Find the largest integral power of two not greater than a value.
bit_width	Find the smallest number of bits needed to represent a value.
countl_zero	Count the number of consecutive bits set to zero, starting from the most significant bit.

Function	Description
countl_one	Count the number of consecutive bits set to one, starting from the most significant bit.
countr_zero	Count the number of consecutive bits set to zero, starting from the least significant bit.
countr_one	Count the number of consecutive bits set to one, starting from the least significant bit.
has_single_bit	Check if a value has only a single bit set to one. This is the same as testing whether a value is a power of two.
popcount	Count the number of bits set to one.
rotl	Compute the result of a bitwise left-rotation.
rotr	Compute the result of a bitwise right-rotation.

See also

Header Files Reference

`<bit>` functions

Article • 12/06/2021

The `<bit>` header includes the following non-member template functions:

Non-member functions	Description
bit_cast	Reinterpret the object representation from one type to another.
bit_ceil	Find the smallest power of two greater than or equal to a value.
bit_floor	Find the largest power of two not greater than a value.
bit_width	Find the smallest number of bits needed to represent a value.
countl_zero	Count the number of consecutive bits set to zero, starting from the most significant bit.
countl_one	Count the number of consecutive bits set to one, starting from the most significant bit.
countr_zero	Count the number of consecutive bits set to zero, starting from the least significant bit.
countr_one	Count the number of consecutive bits set to one, starting from the least significant bit.
has_single_bit	Check if a value has only a single bit set to one. This is the same as testing whether a value is a power of two.
popcount	Count the number of bits set to one.
rotl	Compute the result of a bitwise left rotation.
rotr	Compute the result of a bitwise right rotation.

`bit_cast`

Copy a bit pattern from an object of type `From` to a new object of type `To`.

C++

```
template <class To, class From>
[[nodiscard]] constexpr To bit_cast(const From& from) noexcept;
```

Parameters

To
The type of the output.

From
The type of the value to convert.

from
The value to convert.

Return value

An object of type `To`.

Each bit in the result matches the corresponding bit in `from` unless there are padding bits in `To`, in which case those bits in the result are unspecified.

Example

C++

```
#include <bit>
#include <iostream>

int main()
{
    float f = std::numeric_limits<float>::infinity();
    int i = std::bit_cast<int>(f);
    std::cout << "float f = " << std::hex << f
              << "\nstd::bit_cast<int>(f) = " << std::hex << i << '\n';
    return 0;
}
```

Output

```
float f = inf
std::bit_cast<int>(f) = 7f800000
```

Remarks

Low-level code often needs to interpret an object of one type as another type. The reinterpreted object has the same bit representation as the original, but is a different type.

Instead of using `reinterpret_cast`, or `memcpy()`, `bit_cast()` is a better way to make these conversions. It's better because:

- `bit_cast()` is `constexpr`
- `bit_cast()` requires the types to be trivially copyable and the same size. This prevents potential problems that you could encounter using `reinterpret_cast` and `memcpy` because they could be used to inadvertently, and incorrectly, convert non-trivially-copyable types. Also, `memcpy()` could be used to inadvertently copy between types that aren't the same size. For example, a double (8 bytes) into an unsigned int (4 bytes), or the other way around.

This overload only participates in overload resolution if:

- `sizeof(To) == sizeof(From)`
- `To` and `From` are is_trivially_copyable.

This function template is `constexpr` if and only if `To`, `From`, and the types of their subobjects are:

- Not a union or pointer type
- Not a pointer to member type
- Not volatile-qualified
- Have no non-static data member that is a reference type

`bit_ceil`

Find the smallest power of two greater than or equal to a value. For example, given `3`, returns `4`.

C++

```
template<class T>
[[nodiscard]] constexpr T bit_ceil(T value);
```

Parameters

value
The unsigned integer value to test.

Return value

The smallest power of two greater than or equal to `value`.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    for (auto i = 0u; i < 6u; ++i) // bit_ceil() takes an unsigned integer
type
    {
        auto nextClosestPowerOf2 = std::bit_ceil(i);
        std::cout << "\nbit_ceil(0b" << std::bitset<4>(i) << ") = "
                  << "0b" << std::bitset<4>(nextClosestPowerOf2);
    }
    return 0;
}
```

Output

```
bit_ceil(0b0000) = 0b0001
bit_ceil(0b0001) = 0b0001
bit_ceil(0b0010) = 0b0010
bit_ceil(0b0011) = 0b0100
bit_ceil(0b0100) = 0b0100
bit_ceil(0b0101) = 0b1000
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`bit_floor`

Find the largest power of two not greater than a value. For example, given `5`, returns `4`.

C++

```
template< class T >
[[nodiscard]] constexpr T bit_floor(T value) noexcept;
```

Parameters

value

The unsigned integer value to test.

Return value

The largest power of two that isn't greater than `value`.

If `value` is zero, returns zero.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    for (auto i = 0u; i < 6u; ++i) // bit_floor() takes an unsigned integer
type
    {
        auto previousPowerOf2 = std::bit_floor(i);
        std::cout << "\nbit_floor(0b" << std::bitset<4>(i) << ") = 0b"
                  << std::bitset<4>(previousPowerOf2);
    }
    return 0;
}
```

Output

```
bit_floor(0b0000) = 0b0000
bit_floor(0b0001) = 0b0001
bit_floor(0b0010) = 0b0010
bit_floor(0b0011) = 0b0010
bit_floor(0b0100) = 0b0100
bit_floor(0b0101) = 0b0100
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned`

`char`, and so on.

`bit_width`

Find the smallest number of bits needed to represent a value.

For example, given 5 (0b101), returns 3 because it takes 3 binary bits to express the value 5.

C++

```
template<class T>
[[nodiscard]] constexpr T bit_width(T value) noexcept;
```

Parameters

value

The unsigned integer value to test.

Return value

The number of bits needed to represent `value`.

If `value` is zero, returns zero.

Example

C++

```
#include <bit>
#include <iostream>

int main()
{
    for (unsigned i=0u; i <= 8u; ++i)
    {
        std::cout << "\nbit_width(" << i << ") = "
                  << std::bit_width(i);
    }
    return 0;
}
```

Output

```
bit_width(0) = 0
bit_width(1) = 1
bit_width(2) = 2
bit_width(3) = 2
bit_width(4) = 3
bit_width(5) = 3
bit_width(6) = 3
bit_width(7) = 3
bit_width(8) = 4
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`countl_zero`

Count the number of consecutive bits set to zero, starting from the most significant bit.

C++

```
template<class T>
[[nodiscard]] constexpr int countl_zero(T value) noexcept;
```

Parameters

`value`
The unsigned integer value to test.

Return value

The number of consecutive zero bits, starting from the most significant bit.
If `value` is zero, the number of bits in the type of `value`.

Example

C++

```
#include <bit>
#include <bitset>
```

```cpp
#include <iostream>

int main()
{
    for (unsigned char result = 0, i = 0; i < 9; i++)
    {
        std::cout << "\ncountl_zero(0b" << std::bitset<8>(result) << ") = "
<< std::countl_zero(result);
        result = result == 0 ? 1 : result * 2;
    }
    return 0;
}
```

Output

```
countl_zero(0b00000000) = 8
countl_zero(0b00000001) = 7
countl_zero(0b00000010) = 6
countl_zero(0b00000100) = 5
countl_zero(0b00001000) = 4
countl_zero(0b00010000) = 3
countl_zero(0b00100000) = 2
countl_zero(0b01000000) = 1
countl_zero(0b10000000) = 0
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`countl_one`

Count the number of consecutive bits set to one, starting from the most significant bit.

C++

```cpp
template<class T>
[[nodiscard]] constexpr int countl_one(T value) noexcept;
```

Parameters

`value`

The unsigned integer value to test.

Return value

The number of consecutive bits set to one, starting from the most significant bit.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    unsigned char value = 0;
    for (unsigned char bit = 128; bit > 0; bit /= 2)
    {
        value |= bit;
        std::cout << "\ncountl_one(0b" << std::bitset<8>(value) << ") = "
                  << std::countl_one(value);
    }
    return 0;
}
```

Output

```
countl_one(0b10000000) = 1
countl_one(0b11000000) = 2
countl_one(0b11100000) = 3
countl_one(0b11110000) = 4
countl_one(0b11111000) = 5
countl_one(0b11111100) = 6
countl_one(0b11111110) = 7
countl_one(0b11111111) = 8
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`countr_zero`

Count the number of consecutive bits set to zero, starting from the least significant bit.

C++

```
template<class T>
[[nodiscard]] constexpr int countr_zero(T value) noexcept;
```

Parameters

value

The unsigned integer value to test.

Return value

The number of consecutive zero bits, starting from the least significant bit.
If `value` is zero, the number of bits in the type of `value`.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    for (unsigned char result = 0, i = 0; i < 9; i++)
    {
        std::cout << "\ncountr_zero(0b" << std::bitset<8>(result) << ") = "
                  << std::countr_zero(result);
        result = result == 0 ? 1 : result * 2;
    }
    return 0;
}
```

Output

```
countr_zero(0b00000000) = 8
countr_zero(0b00000001) = 0
countr_zero(0b00000010) = 1
countr_zero(0b00000100) = 2
countr_zero(0b00001000) = 3
countr_zero(0b00010000) = 4
countr_zero(0b00100000) = 5
countr_zero(0b01000000) = 6
countr_zero(0b10000000) = 7
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`countr_one`

Count the number of consecutive bits set to one, starting from the least significant bit.

C++

```
template<class T>
[[nodiscard]] constexpr int countr_one(T value) noexcept;
```

Parameters

`value`
The unsigned integer value to test.

Return value

The number of consecutive bits set to one, starting from the least significant bit.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    unsigned char value = 0;
    for (int bit = 1; bit <= 128; bit *= 2)
    {
        value |= bit;
        std::cout << "\ncountr_one(0b" << std::bitset<8>(value) << ") = "
                  << std::countr_one(value);
    }
    return 0;
}
```

Output

```
countr_one(0b00000001) = 1
countr_one(0b00000011) = 2
countr_one(0b00000111) = 3
countr_one(0b00001111) = 4
countr_one(0b00011111) = 5
countr_one(0b00111111) = 6
countr_one(0b01111111) = 7
countr_one(0b11111111) = 8
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`has_single_bit`

Check if a value has only one bit set.This is the same as testing whether a value is a power of two.

C++

```
template <class T>
[[nodiscard]] constexpr bool has_single_bit(T value) noexcept;
```

Parameters

`value`
The unsigned integer value to test.

Return value

`true` if `value` has only one bit set, which also means that `value` is a power of two. Otherwise, `false`.

Example

C++

```cpp
#include <bit>
#include <bitset>
#include <iostream>
#include <iomanip>

int main()
{
    for (auto i = 0u; i < 10u; ++i)
    {
        std::cout << "has_single_bit(0b" << std::bitset<4>(i) << ") = "
                  << std::boolalpha << std::has_single_bit(i) << '\n';
    }
    return 0;
}
```

Output

```
has_single_bit(0b0000) = false
has_single_bit(0b0001) = true
has_single_bit(0b0010) = true
has_single_bit(0b0011) = false
has_single_bit(0b0100) = true
has_single_bit(0b0101) = false
has_single_bit(0b0110) = false
has_single_bit(0b0111) = false
has_single_bit(0b1000) = true
has_single_bit(0b1001) = false
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`popcount`

Count the number of bits set to one in an unsigned integer value.

C++

```cpp
template<class T>
[[nodiscard]] constexpr int popcount(T value) noexcept;
```

Parameters

value

The unsigned integer value to test.

Return value

The number bits set to one in `value`.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
   for (unsigned char value = 0; value < 16; value++)
    {
        std::cout << "\npopcount(0b" << std::bitset<4>(value) << ") = "
                  << std::popcount(value);
    }
    return 0;
}
```

Output

```
popcount(0b0000) = 0
popcount(0b0001) = 1
popcount(0b0010) = 1
popcount(0b0011) = 2
popcount(0b0100) = 1
popcount(0b0101) = 2
popcount(0b0110) = 2
popcount(0b0111) = 3
popcount(0b1000) = 1
popcount(0b1001) = 2
popcount(0b1010) = 2
popcount(0b1011) = 3
popcount(0b1100) = 2
popcount(0b1101) = 3
popcount(0b1110) = 3
popcount(0b1111) = 4
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`rotl`

Rotates the bits of an unsigned integer value left the specified number of times. Bits that "fall out" of the leftmost bit are rotated into the rightmost bit.

C++

```
template<class T>
[[nodiscard]] constexpr T rotl(T value, int s) noexcept;
```

Parameters

value
The unsigned integer value to rotate.

s
The number of left rotations to perform.

Return value

The result of rotating `value` left, `s` times.
If `s` is zero, returns `value`.
If `s` is negative, does `rotr(value, -s)`. Bits that 'fall out' of the rightmost bit are rotated into the leftmost bit.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    unsigned char bits = 1;
    for (int i = 0; i < 8; ++i)
    {
        std::cout << "rotl(0b" << std::bitset<8>(bits) << ", 1) = ";
```

```
        bits = std::rotl(bits, 1);
        std::cout << "0b" << std::bitset<8>(bits) << '\n';
    }
    std::cout << "rotl(0b" << std::bitset<8>(bits) << ",-1) = ";
    bits = std::rotl(bits, -1);
    std::cout << "0b" << std::bitset<8>(bits);
    return 0;
}
```

Output

```
rotl(0b00000001, 1) = 0b00000010
rotl(0b00000010, 1) = 0b00000100
rotl(0b00000100, 1) = 0b00001000
rotl(0b00001000, 1) = 0b00010000
rotl(0b00010000, 1) = 0b00100000
rotl(0b00100000, 1) = 0b01000000
rotl(0b01000000, 1) = 0b10000000
rotl(0b10000000, 1) = 0b00000001
rotl(0b00000001,-1) = 0b10000000
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned char`, and so on.

`rotr`

Rotates the bits of `value` right the specified number of times. Bits that 'fall out' of the rightmost bit are rotated back into the leftmost bit.

C++

```
template<class T>
[[nodiscard]] constexpr T rotr(T value, int s) noexcept;
```

Parameters

`value`

The unsigned integer value to rotate.

`s`

The number of right rotations to perform.

Return value

The result of rotating `value` right, `s` times.
If `s` is zero, returns `value`.
If `s` is negative, does `rotl(value, -s)`. Bits that 'fall out' of the leftmost bit are rotated back into the rightmost bit.

Example

C++

```
#include <bit>
#include <bitset>
#include <iostream>

int main()
{
    unsigned char bits = 128;
    for (int i = 0; i < 8; ++i)
    {
        std::cout << "rotr(0b" << std::bitset<8>(bits) << ", 1) = ";
        bits = std::rotr(bits, 1);
        std::cout << "0b" << std::bitset<8>(bits) << '\n';
    }
    std::cout << "rotr(0b" << std::bitset<8>(bits) << ",-1) = ";
    bits = std::rotr(bits, -1);
    std::cout << "0b" << std::bitset<8>(bits);
    return 0;
}
```

Output

```
rotr(0b10000000, 1) = 0b01000000
rotr(0b01000000, 1) = 0b00100000
rotr(0b00100000, 1) = 0b00010000
rotr(0b00010000, 1) = 0b00001000
rotr(0b00001000, 1) = 0b00000100
rotr(0b00000100, 1) = 0b00000010
rotr(0b00000010, 1) = 0b00000001
rotr(0b00000001, 1) = 0b10000000
rotr(0b10000000,-1) = 0b00000001
```

Remarks

This template function only participates in overload resolution if `T` is an unsigned integer type. For example: `unsigned int`, `unsigned long`, `unsigned short`, `unsigned`

`char`, and so on.

Requirements

Header: `<bit>`

Namespace: std

/std:c++20 or later is required.

See also

<bit>

`endian` enum

Article • 09/23/2021

Indicates the endianness of all scalar types.

Syntax

```
C++
```

```
enum class endian {
    little = 0,
    big = 1,
    native = little
 };
```

Members

Element	Description
`little`	Indicates that scalar types are little-endian. That is, the least significant byte is stored in the smallest address. For example, `0x1234` is stored `0x34` `0x12`.
`big`	Indicates that scalar types are big-endian, that is, the most significant byte is stored in the smallest address. For example, `0x1234` is stored `0x12` `0x34`.

Remarks

All native scalar types are little-endian for the platforms that Microsoft Visual C++ targets (x86, x64, ARM, ARM64).

Requirements

Header: `<bit>`

Namespace: `std`

Compiler option: /std:c++20 or later is required.

See also

<bit>

<bitset>

Article • 05/20/2022

Defines the class template bitset and two supporting template functions for representing and manipulating fixed-size sequences of bits.

Requirements

Header: <bitset>

Namespace: std

Members

Operators

Name	Description
operator&	Performs a bitwise AND between two bitsets.
operator<<	Inserts a text representation of the bit sequence into the standard output stream.
operator>>	Inserts a text representation of the bit sequence into the standard input stream.
operator^	Performs a bitwise EXCLUSIVE-OR between two bitsets.
operator|	Performs a bitwise OR between two bitsets.

Classes

Name	Description
bitset	The class template describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions.

See also

Header Files Reference
Thread Safety in the C++ Standard Library

`<bitset>` operators

Article • 05/20/2022

operator&

Performs a bitwise `AND` between two bitsets.

```
C++
```

```
template <size_t size>
bitset<size>
operator&(
    const bitset<size>& left,
    const bitset<size>& right);
```

Parameters

left
The first of the two bitsets whose respective elements are to be combined with the bitwise `AND`.

right
The second of the two valarrays whose respective elements are to be combined with the bitwise `AND`.

Return Value

A bitset whose elements are the result of performing the `AND` operation on the corresponding elements of *left* and *right*.

Example

```
C++
```

```
// bitset_and.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 & b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0001
```

`operator<<`

Inserts a text representation of the bit sequence into the output stream.

C++

```
template <class CharType, class Traits, size_t N>
basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& ostr,
    const bitset<N>& right);
```

Parameters

right
An object of type **bitset<N>** that is to be inserted into the output stream as a string.

Return Value

A text representation of the bit sequence in `ostr`.

Remarks

The template function overloads `operator<<`, allowing a bitset to be written out without first converting it into a string. The template function effectively executes:

`ostr << right.`to_string`<CharType, Traits, allocator<CharType>>()`

Example

C++

```
// bitset_op_insert.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 9 );

   // bitset inserted into output stream directly
   cout << "The ordered set of bits in the bitset<5> b1(9)"
        << "\n can be output with the overloaded << as: ( "
        << b1 << " )" << endl;

   // Compare converting bitset to a string before
   // inserting it into the output steam
   string s1;
   s1 =  b1.template to_string<char,
      char_traits<char>, allocator<char> >( );
   cout << "The string returned from the bitset b1"
        << "\n by the member function to_string( ) is: "
        << s1 << "." << endl;
}
```

`operator>>`

Reads a string of bit characters into a bitset.

C++

```
template <class CharType, class Traits, size_t Bits>
basic_istream<CharType, Traits>& operator>> (
    basic_istream<CharType, Traits>& i_str,
    bitset<N>& right);
```

Parameters

i_str

The string that is entered into the input stream to be inserted into the bitset.

right
The bitset that is receiving the bits from the input stream.

Return Value

The template function returns the string *i_str*.

Remarks

The template function overloads `operator>>` to store in the bitset *right* the value `bitset(str)`, where `str` is an object of type basic_string`< CharType, Traits, allocator< CharType > >&` extracted from *i_str*.

The template function extracts elements from *i_str* and inserts them into the bitset until:

- All the bit elements have been extracted from the input stream and stored in the bitset.
- The bitset is filled up with bits from the input stream.
- An input element is encountered that is neither 0 nor 1.

Example

C++

```
#include <bitset>
#include <iostream>
#include <string>

using namespace std;
int main()
{

   bitset<5> b1;
   cout << "Enter string of (0 or 1) bits for input into bitset<5>.\n"
        << "Try bit string of length less than or equal to 5,\n"
        << " (for example: 10110): ";
   cin >>  b1;

   cout << "The ordered set of bits entered from the "
        << "keyboard\n has been input into bitset<5> b1 as: ( "
        << b1 << " )" << endl;

   // Truncation due to longer string of bits than length of bitset
   bitset<2> b3;
   cout << "Enter string of bits (0 or 1) for input into bitset<2>.\n"
```

```
         << " Try bit string of length greater than 2,\n"
         << " (for example: 1011): ";
    cin >>  b3;

    cout << "The ordered set of bits entered from the "
         << "keyboard\n has been input into bitset<2> b3 as: ( "
         << b3 << " )" << endl;

    // Flushing the input stream
    char buf[100];
    cin.getline(&buf[0], 99);

    // Truncation with non-bit value
    bitset<5> b2;
    cout << "Enter a string for input into  bitset<5>.\n"
         << " that contains a character than is NOT a 0 or a 1,\n "
         << " (for example: 10k01): ";
    cin >>  b2;

    cout << "The string entered from the keyboard\n"
         << " has been input into bitset<5> b2 as: ( "
         << b2 << " )" << endl;
}
```

operator^

Performs a bitwise `EXCLUSIVE-OR` between two bitsets.

C++

```
template <size_t size>
bitset<size>
operator^(
    const bitset<size>& left,
    const bitset<size>& right);
```

Parameters

left
The first of the two bitsets whose respective elements are to be combined with the bitwise `EXCLUSIVE-OR`.

right
The second of the two valarrays whose respective elements are to be combined with the bitwise `EXCLUSIVE-OR`.

Return Value

A bitset whose elements are the result of performing the `EXCLUSIVE-OR` operation on the corresponding elements of *left* and *right*.

Example

C++

```
// bitset_xor.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 ^ b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0110
```

`operator|`

Performs a bitwise OR between two `bitset` objects.

C++

```
template <size_t size>
bitset<size>
operator|(
    const bitset<size>& left,
    const bitset<size>& right);
```

Parameters

left
The first of the two bitsets whose respective elements are to be combined with the bitwise `OR`.

right
The second of the two valarrays whose respective elements are to be combined with the bitwise `OR`.

Return Value

A bitset whose elements are the result of performing the `OR` operation on the corresponding elements of *left* and *right*.

Example

C++

```
// bitset_or.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

using namespace std;

int main()
{
   bitset<4> b1 ( string("0101") );
   bitset<4> b2 ( string("0011") );
   bitset<4> b3 = b1 | b2;
   cout << "bitset 1: " << b1 << endl;
   cout << "bitset 2: " << b2 << endl;
   cout << "bitset 3: " << b3 << endl;
}
```

Output

```
bitset 1: 0101
bitset 2: 0011
bitset 3: 0111
```

`bitset` class

Article • 06/16/2022

Describes a type of object that stores a sequence consisting of a fixed number of bits that provide a compact way of keeping flags for a set of items or conditions. The `bitset` class supports operations on objects of type bitset that contain a collection of bits and provide constant-time access to each bit.

Syntax

```
C++
```

```
template <size_t N>
class bitset
```

Parameters

`N`

Specifies the number of bits in the `bitset` object with a nonzero integer of type `size_t` that must be known at compile time.

Remarks

Unlike the similar vector<bool> Class, the `bitset` class doesn't have iterators and isn't a C++ Standard Library container. It also differs from `vector<bool>` by being of some specific size that is fixed at compile time in accordance with the size specified by the template parameter *`N`* when the **`bitset<N>`** is declared.

A bit is set if its value is 1 and reset if its value is 0. To flip or invert a bit is to change its value from 1 to 0 or from 0 to 1. The *`N`* bits in a `bitset` are indexed by integer values from 0 to *`N`* - 1, where 0 indexes the first bit position and *`N`* - 1 the final bit position.

Members

Constructors

Name	Description

Name	Description
bitset	Constructs an object of class `bitset<N>` and initializes the bits to zero, to some specified value, or to values obtained from characters in a string.

Typedefs

Name	Description
element_type	A type that is a synonym for the data type **`bool`** and can be used to reference element bits in a `bitset`.

Functions

Name	Description
all	Tests all of the bits in this `bitset` to determine whether they're all set to **`true`**.
any	The member function tests whether any bit in the sequence is set to 1.
count	The member function returns the number of bits set in the bit sequence.
flip	Inverts the value of all the bits in a `bitset` or inverts a single bit at a specified position.
none	Tests if no bit has been set to 1 in a `bitset` object.
reset	Resets all the bits in a `bitset` to 0 or resets a bit at a specified position to 0.
set	Sets all the bits in a `bitset` to 1 or sets a bit at a specified position to 1.
size	Returns the number of bits in a `bitset` object.
test	Tests whether the bit at a specified position in a `bitset` is set to 1.
to_string	Converts a `bitset` object to a string representation.
to_ullong	Returns the sum of the bit values in the `bitset` as an **`unsigned long long`**.
to_ulong	Converts a `bitset` object to the **`unsigned long`** that would generate the sequence of bits contained if used to initialize the `bitset`.

Classes

Name	Description

Name	Description
reference	A proxy class that provides references to bits contained in a `bitset` that is used to access and manipulate the individual bits as a helper class for the `operator[]` of class `bitset`.

Operators

Name	Description
operator!=	Tests a target `bitset` for inequality with a specified `bitset`.
operator&=	Does a bitwise combination of bitsets with the bitwise "and" (`&`) operation.
operator<<	Shifts the bits in a `bitset` to the left a specified number of positions and returns the result to a new `bitset`.
operator<<=	Shifts the bits in a `bitset` to the left a specified number of positions and returns the result to the targeted `bitset`.
operator==	Tests a target `bitset` for equality with a specified `bitset`.
operator>>	Shifts the bits in a `bitset` to the right a specified number of positions and returns the result to a new `bitset`.
operator>>=	Shifts the bits in a `bitset` to the right a specified number of positions and returns the result to the targeted `bitset`.
operator[]	Returns a reference to a bit at a specified position in a `bitset` if the `bitset` is modifiable; otherwise, it returns the value of the bit at that position.
operator^=	Does a bitwise combination of bitsets with the bitwise "xor" (`^`) operation.
operator|=	Does a bitwise combination of bitsets with the bitwise "or" (`|`) operation.
operator~	Inverts all the bits in a target `bitset` and returns the result.

Structures

Name	Description
hash	

`all`

Tests all of the bits in this bitset to determine if they're all set to true.

C++

```
bool all() const;
```

Return Value

Returns `true` if all bits in this set are true. Returns **`false`** if one or more bits are false.

any

Tests whether any bit in the sequence is set to 1.

C++

```
bool any() const;
```

Return Value

`true` if any bit in this `bitset` are set to 1; **`false`** if all the bits are 0.

Example

C++

```
// bitset_any.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   bool b, rb;

   cout << "The original bitset b1( 6 ) is: ( "<< b1 << " )"
        << endl;

   b = b1.any ( );

   if ( b )
      cout << "At least one of the bits in bitset is set to 1."
           << endl;
   else
```

```
      cout << "None of the bits in bitset are set to 1."
           << endl;

   bitset<5> rb1;
   rb1 = b1.reset ( );

   cout << "The reset bitset is: ( "<< b1 << " )"
        << endl;

   rb = rb1.any ( );

   if ( rb )
      cout << "At least one of the bits in the reset bitset "
           << "are set to 1." << endl;
   else
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
}
```

Output

```
The original bitset b1( 6 ) is: ( 00110 )
At least one of the bits in bitset is set to 1.
The reset bitset is: ( 00000 )
None of the bits in bitset b1 are set to 1.
```

bitset

Constructs an object of class `bitset<N>` and initializes the bits to zero, or to some specified value, or to values obtained from characters in a string.

C++

```
1) constexpr bitset();
2) bitset(unsigned long val);
3) constexpr bitset(unsigned long long val);
4) template <class CharType, class Traits, class Allocator>
    explicit bitset(
        const basic_string<CharType, Traits, Allocator>& str,
        typename basic_string<CharType, Traits, Allocator>::size_type pos =
0);

5) template <class CharType, class Traits, class Allocator>
    explicit bitset(
        const basic_string<CharType, Traits, Allocator>& str,
        typename basic_string<CharType, Traits, Allocator>::size_type pos,
        typename basic_string<CharType, Traits, Allocator>::size_type count,
        CharType Zero = CharType ('0'),
        CharType One = CharType ('1'));
```

```
6) template<class CharType>
    explicit bitset(
        const CharType* str,
        typename basic_string<CharType>::size_type
        n = basic_string<CharType>::npos,
        CharType zero = CharType('0'),
        CharType one = CharType('1'));
```

Parameters

val

The unsigned integer whose base two representation is used to initialize the bits in the `bitset` being constructed.

str

The string of zeros and ones used to initialize the `bitset` bit values.

pos

The position of the character in the string, counting from left to right and starting with zero, used to initialize the first bit in the `bitset`.

count

The number of characters in the string that is used to provide initial values for the bits in the `bitset`.

Zero

The character that is used to represent a zero. The default is '0'.

One

The character that is used to represent a one. The default is '1'.

Remarks

`1)` Constructs an object of class `bitset<N>` and initializes all N bits to a default value of zero.

`2-3)` Constructs an object of class `bitset<N>` and initializes the bits from the `val` parameter.

`4)` Constructs an object of class `bitset<N>` and initializes bits from the characters provided in a string of zeros and ones. If any characters of the string are other than 0 or 1, the constructor throws an object of class invalid argument. If the position specified (*pos*) is beyond the length of the string, then the constructor throws an object of class

out_of_range. The constructor sets only those bits at position *j* in the `bitset` for which the character in the string at position `pos + j` is 1. By default, *`pos`* is 0.

`5)` Similar to `4)` but includes another parameter, *`count`*, that specifies the number of bits to initialize. It has two optional parameters, *`_Zero`* and *`_One`*, which indicate what character in *`str`* should be interpreted to mean a 0 bit and a 1 bit, respectively.

`6)` Constructs an object of class `bitset<N>`, initializing the N bits to values that correspond to the characters provided in a C-style character string of zeros and ones. You call the constructor without casting the string into a string type, for example: `bitset<5> b5("01011");`

Example

C++

```
// bitset_bitset.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   // Using the default constructor
   using namespace std;
   bitset<2> b0;
   cout << "The set of bits in bitset<2> b0 is: ( "
        << b0 << " )." << endl;

   // Using the second member function
   bitset<5> b1 ( 6 );
   cout << "The set of bits in bitset<5> b1( 6 ) is: ( "
        << b1 << " )." << endl;

   // The template parameter N can be an expresssion
   bitset< 2 * sizeof ( int ) > b2;
   cout << "The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( "
        << b2 << " )." << endl;

   // The base two representation will be truncated
   // if its length exceeds the size of the bitset
   bitset<3> b3 ( 6 );
   cout << "The set of bits in bitset<3> b3( 6 ) is ( "
        << b3 << " )." << endl;

   // Using a c-style string to initialize the bitset
    bitset<7> b3andahalf ( "1001001" );
    cout << "The set of bits in bitset<7> b3andahalf ( \"1001001\" )"
         << " is ( " << b3andahalf << " )." << endl;
```

```
    // Using the fifth member function with the first parameter
    string bitval4 ( "10011" );
    bitset<5> b4 ( bitval4 );
    cout << "The set of bits in bitset<5> b4( bitval4 ) is ( "
         << b4 << " )." << endl;

    // Only part of the string may be used for initialization

    // Starting at position 3 for a length of 6 (100110)
    string bitval5 ("11110011011");
    bitset<6> b5 ( bitval5, 3, 6 );
    cout << "The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( "
         << b5 << " )." << endl;

    // The bits not initialized with part of the string
    // will default to zero
    bitset<11> b6 ( bitval5, 3, 5 );
    cout << "The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( "
         << b6 << " )." << endl;

    // Starting at position 2 and continue to the end of the string
    bitset<9> b7 ( bitval5, 2 );
    cout << "The set of bits in bitset<9> b7( bitval, 2 ) is ( "
         << b7 << " )." << endl;
}
```

Output

```
The set of bits in bitset<2> b0 is: ( 00 ).
The set of bits in bitset<5> b1( 6 ) is: ( 00110 ).
The set of bits in bitset<2 * sizeof ( int ) > b2 is: ( 00000000 ).
The set of bits in bitset<3> b3( 6 ) is ( 110 ).
The set of bits in bitset<5> b4( bitval4 ) is ( 10011 ).
The set of bits in bitset<11> b5( bitval, 3, 6 ) is ( 100110 ).
The set of bits in bitset<11> b6( bitval5, 3, 5 ) is ( 00000010011 ).
The set of bits in bitset<9> b7( bitval, 2 ) is ( 110011011 ).
```

count

Returns the number of bits set in the bit sequence.

C++

```
size_t count() const;
```

Return Value

The number of bits set in the bit sequence.

Example

C++

```
// bitset_count.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
    using namespace std;

    bitset<5> b1(4);

    cout << "The collection of bits in the original bitset is: ( "
         << b1 << " )" << endl;

    size_t i;
    i = b1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << i << "." << endl;

    bitset<5> fb1;
    fb1 = b1.flip();

    cout << "The collection of flipped bits in the modified bitset "
         << "is: ( " << b1 << " )" << endl;

    size_t ii;
    ii = fb1.count();
    cout << "The number of bits in the bitset set to 1 is: "
         << ii << "." << endl;
}
```

Output

```
The collection of bits in the original bitset is: ( 00100 )
The number of bits in the bitset set to 1 is: 1.
The collection of flipped bits in the modified bitset is: ( 11011 )
The number of bits in the bitset set to 1 is: 4.
```

`element_type`

A type that is a synonym for the data type **`bool`** and can be used to reference element bits in a `bitset`.

C++

```
typedef bool element_type;
```

Example

C++

```
// bitset_elem_type.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<3> b1 ( 2 );
   cout << "Original bitset b1(6) is: ( "<< b1 << " )"
        << endl;

   //Compare two ways to reference bits in a bitset
   bool b;
   bitset<5>::element_type e;

   b = b1.test ( 2 );
   if ( b )
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 0." << endl;
   b1[2] = 1;
   cout << "Bitset b1 modified by b1[2] = 1 is: ( "<< b1 << " )"
        << endl;

   e = b1.test ( 2 );
   if ( e )
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 1." << endl;
   else
      cout << "The bit at position 2 of bitset b1"
           << "has a value of 0." << endl;
}
```

Output

```
Original bitset b1(6) is: ( 010 )
The bit at position 2 of bitset b1has a value of 0.
```

```
Bitset b1 modified by b1[2] = 1 is: ( 110 )
The bit at position 2 of bitset b1has a value of 1.
```

`flip`

Inverts the value of all the bits in a `bitset` or inverts a single bit at a specified position.

C++

```
bitset<N>& flip();
bitset<N>& flip(size_t pos);
```

Parameters

pos

The position of the bit whose value is to be inverted.

Return Value

A copy of the modified `bitset` for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified as a parameter is greater than the size *N* of the **bitset<N>** whose bit was inverted.

Example

C++

```
// bitset_flip.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 6 );

   cout << "The collection of bits in the original bitset is: ( "
        << b1 << " )" << endl;

   bitset<5> fb1;
```

```
   fb1 = b1.flip ( );

   cout << "After flipping all the bits, the bitset becomes: ( "
        << fb1 << " )" << endl;

   bitset<5> f3b1;
   f3b1 = b1.flip ( 3 );

   cout << "After flipping the fourth bit, the bitset becomes: ( "
        << f3b1 << " )" << endl << endl;

   bitset<5> b2;
   int i;
   for ( i = 0 ; i <= 4 ; i++ )
   {
      b2.flip(i);
      cout << b2 << "  The bit flipped is in position "
           << i << ".\n";
   }
}
```

Output

```
The collection of bits in the original bitset is: ( 00110 )
After flipping all the bits, the bitset becomes: ( 11001 )
After flipping the fourth bit, the bitset becomes: ( 10001 )

00001  The bit flipped is in position 0.
00011  The bit flipped is in position 1.
00111  The bit flipped is in position 2.
01111  The bit flipped is in position 3.
11111  The bit flipped is in position 4.
```

hash

C++

```
template <class T> struct hash;
template <size_t N> struct hash<bitset<N>>;
```

none

Tests if no bit has been set to 1 in a `bitset` object.

C++

```
bool none() const;
```

Return Value

`true` if no bit in the `bitset` has been set to 1; `false` if at least one bit has been set to 1.

Example

C++

```
// bitset_none.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 6 );
   bool b, rb;

   cout << "Original bitset b1(6) is: ( " << b1 << " )"
        << endl;

   b = b1.none ( );

   if ( b )
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
   else
      cout << "At least one of the bits in bitset b1 is set to 1."
           << endl;

   bitset<5> rb1;
   rb1 = b1.reset ( );
   rb = rb1.none ( );
   if ( rb )
      cout << "None of the bits in bitset b1 are set to 1."
           << endl;
   else
      cout << "At least one of the bits in bitset b1 is set to 1."
           << endl;
}
```

Output

```
Original bitset b1(6) is: ( 00110 )
At least one of the bits in bitset b1 is set to 1.
None of the bits in bitset b1 are set to 1.
```

`operator!=`

Tests a target bitset for inequality with a specified bitset.

C++

```
bool operator!=(const bitset<N>& right) const;
```

Parameters

`right`

The `bitset` that is to be compared to the target bitset for inequality.

Return Value

`true` if the bitsets are different; **`false`** if they're the same.

Remarks

Bitsets must be the same size.

Example

C++

```
// bitset_op_NE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 7 );
   bitset<5> b3 ( 2 );
   bitset<4> b4 ( 7 );

   if ( b1 != b2 )
      cout << "Bitset b1 is different from bitset b2." << endl;
   else
      cout << "Bitset b1 is the same as bitset b2." << endl;

   if ( b1 != b3 )
      cout << "Bitset b1 is different from bitset b3." << endl;
```

```
   else
      cout << "Bitset b1 is the same as bitset b3." << endl;

   // This would cause an error because bitsets must have the
   // same size to be tested
   // if ( b1 != b4 )
   //   cout << "Bitset b1 is different from bitset b4." << endl;
   // else
   //   cout << "Bitset b1 is the same as bitset b4." << endl;
}
```

Output

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

`operator&=`

Does a bitwise combination of bitsets with the bitwise "and" (`&`) operation.

C++

```
bitset<N>& operator&=(const bitset<N>& right);
```

Parameters

`right`

The `bitset` that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise "and" (`&`) operation with the `bitset` specified as a parameter.

Remarks

Two bits combined by the `AND` operator return **`true`** if each bit is true; otherwise, their combination returns **`false`**.

The two bitsets must be the same size.

Example

C++

```
// bitset_op_bitwise.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 &= b2;
   cout << "After bitwise AND combination,\n"
        << "the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

   // Note that the parameter-specified bitset is unchanged
   cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

   // The following would cause an error because the bisets
   // must be of the same size to be combined
   // b1 &= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise AND combination,
the target bitset b1 becomes:   ( 00011 ).
The parameter bitset b2 remains: ( 01011 ).
```

`operator<<`

Shifts the bits in a `bitset` to the left a specified number of positions and returns the result to a new `bitset`.

C++

```
bitset<N> operator<<(size_t pos) const;
```

Parameters

pos

The number of positions to the left that the bits in the `bitset` are to be shifted.

Return Value

The modified bitset with the bits shifted to the left the required number of positions.

Remarks

The member operator function returns **`bitset(*this) <<= pos`** where `<<=` shifts the bits in a `bitset` to the left a specified number of positions and returns the result to the targeted `bitset`.

Example

C++

```
// bitset_op_LS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

   bitset<5> b2;
   b2 = b1 << 2;

   cout << "After shifting the bits 2 positions to the left,\n"
        << " the bitset b2 is: ( "<< b2 << " )."
        << endl;

   bitset<5> b3 = b2 >> 1;

   cout << "After shifting the bits 1 position to the right,\n"
        << " the bitset b3 is: ( " << b3 << " )."
```

```
        << endl;
}
```

`operator<<=`

Shifts the bits in a `bitset` to the left a specified number of positions and returns the result to the targeted `bitset`.

C++

```
bitset<N>& operator<<=(size_t pos);
```

Parameters

pos
The number of positions to the left the bits in the `bitset` are to be shifted.

Return Value

The targeted `bitset` modified so that the bits have been shifted to the left the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

C++

```
// bitset_op_LSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;
   b1 <<= 2;
   cout << "After shifting the bits 2 positions to the left,\n"
        << "the target bitset b1 becomes: ( "<< b1 << " )."
```

```
        << endl;
}
```

Output

```
The target bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
the target bitset b1 becomes: ( 11100 ).
```

`operator==`

Tests a target bitset for equality with a specified bitset.

C++

```
bool operator==(const bitset<N>& right) const;
```

Parameters

`right`

The `bitset` that is to be compared to the target bitset for equality.

Return Value

`true` if the bitsets are the same; **`false`** if they're different.

Remarks

Bitsets must be the same size.

Example

C++

```
// bitset_op_EQ.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
```

```
   bitset<5> b2 ( 7 );
   bitset<5> b3 ( 2 );
   bitset<4> b4 ( 7 );

   if ( b1 == b2 )
      cout << "Bitset b1 is the same as bitset b2." << endl;
   else
      cout << "Bitset b1 is different from bitset b2." << endl;

   if ( b1 == b3 )
      cout << "Bitset b1 is the same as bitset b3." << endl;
   else
      cout << "Bitset b1 is different from bitset b3." << endl;

   // This would cause an error because bitsets must have the
   // same size to be tested
   // if ( b1 == b4 )
   //   cout << "Bitset b1 is the same as bitset b4." << endl;
   // else
   //   cout << "Bitset b1 is different from bitset b4." << endl;
}
```

Output

```
Bitset b1 is the same as bitset b2.
Bitset b1 is different from bitset b3.
```

`operator>>`

Shifts the bits in a `bitset` to the right a specified number of positions and returns the result to a new bitset.

C++

```
bitset<N> operator>>(size_t pos) const;
```

Parameters

pos

The number of positions to the right the bits in the `bitset` are to be shifted.

Return Value

A new bitset where the bits have been shifted to the right the required number of positions relative to the targeted `bitset`.

Example

C++

```
// bitset_op_RS.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   cout << "The bitset b1 is: ( "<< b1 << " )." << endl;

   bitset<5> b2;
   b2 = b1 << 2;

   cout << "After shifting the bits 2 positions to the left,\n"
        << "the bitset b2 is: ( "<< b2 << " )."
        << endl;
   bitset<5> b3 = b2 >> 1;

   cout << "After shifting the bits 1 position to the right,\n"
        << "the bitset b3 is: ( " << b3 << " )."
        << endl;
}
```

Output

```
The bitset b1 is: ( 00111 ).
After shifting the bits 2 positions to the left,
the bitset b2 is: ( 11100 ).
After shifting the bits 1 position to the right,
the bitset b3 is: ( 01110 ).
```

`operator>>=`

Shifts the bits in a `bitset` to the right a specified number of positions and returns the result to the targeted `bitset`.

C++

```
bitset<N>& operator>>=(size_t pos);
```

Parameters

pos

The number of positions to the right the bits in the `bitset` are to be shifted.

Return Value

The targeted `bitset` modified so that the bits have been shifted to the right the required number of positions.

Remarks

If no element exists to shift into the position, the function clears the bit to a value of 0.

Example

C++

```
// bitset_op_RSE.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 28 );
   cout << "The target bitset b1 is: ( "<< b1 << " )." << endl;

   b1 >>= 2;
   cout << "After shifting the bits 2 positions to the right,\n"
        << "the target bitset b1 becomes: ( "<< b1 << " )."
        << endl;
}
```

Output

```
The target bitset b1 is: ( 11100 ).
After shifting the bits 2 positions to the right,
the target bitset b1 becomes: ( 00111 ).
```

`operator[]`

Returns a reference to a bit at a specified position in a `bitset` if the `bitset` is modifiable; otherwise, it returns the value of the bit at that position.

C++

```
bool operator[](size_t pos) const;
reference operator[](size_t pos);
```

Parameters

pos

The position locating the bit within the `bitset`.

Remarks

When you define _ITERATOR_DEBUG_LEVEL as 1 or 2 in your build, a runtime error will occur in your executable if you attempt to access an element outside the bounds of the `bitset`. For more information, see Checked Iterators.

Example

C++

```
// bitset_op_REF.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bool b;
   bitset<5> b1 ( 6 );
   cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
        << endl;

   int i;
   for ( i = 0 ; i <= 4 ; i++ )
   {
      b = b1[ i ];
      cout << "  The bit in position "
           << i << " is " << b << ".\n";
   }
}
```

`operator^=`

Does a bitwise combination of bitsets with the bitwise "xor" (`^`) operation.

C++

```
bitset<N>& operator^=(const bitset<N>& right);
```

Parameters

right

The `bitset` that is to be combined bitwise with the target bitset.

Return Value

The modified target bitset that results from the bitwise "xor" (`^`) operation with the `bitset` specified as a parameter.

Remarks

Two bits combined by the bitwise "xor" operator (`^`) return **`true`** if at least one, but not both, of the bits are **`true`**; otherwise, their combination returns **`false`**.

Bitsets must be the same size.

Example

C++

```
// bitset_op_bitwiseOR.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;
   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 ^= b2;
   cout << "After bitwise exclusive OR combination,\n"
        << "the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;
```

```
    // Note that the parameter-specified bitset in unchanged
    cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
         << endl;

    // The following would cause an error because the bitsets
    // must be of the same size to be combined
    // b1 |= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise exclusive OR combination,
the target bitset b1 becomes:   ( 01100 ).
The parameter bitset b2 remains: ( 01011 ).
```

`operator|=`

Combines two bitsets using the bitwise "or" (`|`) operation.

C++

```
bitset<N>& operator|=(const bitset<N>& right);
```

Parameters

`right`

The `bitset` that is to be combined bitwise with the target `bitset`.

Return Value

The modified target bitset that results from the bitwise "or" (`|`) operation with the `bitset` specified as a parameter.

Remarks

Two bits combined by the inclusive `OR` operator return **`true`** if at least one of the bits is **`true`**; if both bits are **`false`**, their combination returns **`false`**.

Bitsets must be the same size.

Example

C++

```
// bitset_op_BIO.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2 ( 11 );
   bitset<4> b3 ( 7 );

   cout << "The target bitset b1 is:    ( "<< b1 << " )." << endl;
   cout << "The parameter bitset b2 is: ( "<< b2 << " )." << endl;
   cout << endl;

   b1 |= b2;
   cout << "After bitwise inclusive OR combination,\n"
        << "the target bitset b1 becomes:   ( "<< b1 << " )."
        << endl;

   // Note that the parameter-specified bitset in unchanged
   cout << "The parameter bitset b2 remains: ( "<< b2 << " )."
        << endl;

   // The following would cause an error because the bisets
   // must be of the same size to be combined
   // b1 |= b3;
}
```

Output

```
The target bitset b1 is:    ( 00111 ).
The parameter bitset b2 is: ( 01011 ).

After bitwise inclusive OR combination,
the target bitset b1 becomes:   ( 01111 ).
The parameter bitset b2 remains: ( 01011 ).
```

`operator~`

Inverts all the bits in a target bitset and returns the result.

C++

```
bitset<N> operator~() const;
```

Return Value

The `bitset` with all its bits inverted with respect to the targeted `bitset`.

Example

C++

```
// bitset_op_invert.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <bitset>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );
   bitset<5> b2;
   b2 = ~b1;

   cout << "Bitset b1 is: ( "<< b1 << " )." << endl;
   cout << "Bitset b2 = ~b1 is: ( "<< b2 << " )." << endl;

   // These bits could also be flipped using the flip member function
   bitset<5> b3;
   b3 = b1.flip( );
   cout << "Bitset b3 = b1.flip( ) is: ( "<< b2 << " )." << endl;
}
```

Output

```
Bitset b1 is: ( 00111 ).
Bitset b2 = ~b1 is: ( 11000 ).
Bitset b3 = b1.flip( ) is: ( 11000 ).
```

`reference`

A proxy class that provides references to bits contained in a `bitset` that is used to access and manipulate the individual bits as a helper class for the `operator[]` of class `bitset`.

C++

```
class reference {
   friend class bitset<N>;
public:
   reference& operator=(bool val);
   reference& operator=(const reference& bitref);
   bool operator~() const;
   operator bool() const;
   reference& flip();
};
```

Parameters

val

The value of the object of type **bool** to be assigned to a bit in a bitset.

bitref

A reference of the form *x [i]* to the bit at position *i* in bitset *x*.

Return Value

A reference to the bit in the bitset specified by the argument position for the first, second, and fifth member functions of class reference, and **true** or **false**, to reflect the value of the modified bit in the bitset for the third and fourth member functions of class reference.

Remarks

The class reference exists only as a helper class for the bitset operator[]. The member class describes an object that can access an individual bit within a bitset. Let *b* be an object of type **bool**, *x* and *y* objects of type **bitset<N>**, and *i* and *j* valid positions within such an object. The notation *x [i]* references the bit at position *i* in bitset *x*. The member functions of class reference provide, in order, the following operations:

Operation	Definition
x[*i*] = *b*	Stores **bool** value *b* at bit position *i* in bitset *x*.
x[*i*] = *y*[*j*]	Stores the value of the bit *y*[*j*] at bit position *i* in bitset *x*.
b = ~ *x*[*i*]	Stores the flipped value of the bit *x*[*i*] in **bool** *b*.

Operation	Definition
`b = x[i]`	Stores the value of the bit `x[i]` in `bool b`.
`x[i].flip()`	Stores the flipped value of the bit `x[i]` back at bit position `i` in `x`.

Example

C++

```
// bitset_reference.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 2 );
   bitset<5> b2 ( 6 );
   cout << "The initialized bitset<5> b1( 2 ) is: ( "<< b1 << " )."
        << endl;
   cout << "The initialized bitset<5> b2( 6 ) is: ( "<< b2 << " )."
        << endl;

   // Example of x [i] = b storing bool b at bit position i
   // in bitset x
   b1[ 0 ] = true;
   cout << "The bitset<5> b1 with the bit at position 0 set to 1"
        << "is: ( "<< b1 << " )" << endl;

   // Example of x [i] = y [j] storing the bool value of the
   // bit at position j in bitset y at bit position i in bitset x
   b2 [4] = b1 [0];      // b1 [0] = true
   cout << "The bitset<5> b2 with the bit at position 4 set to the "
        << "value\nof the bit at position 0 of the bit in "
        << "bitset<5> b1 is: ( "<<  b2  << " )" << endl;

   // Example of b = ~x [i] flipping the value of the bit at
   // position i of bitset x and storing the value in an
   // object b of type bool
   bool b = ~b2 [4];      // b2 [4] = false
   if ( b )
      cout << "The value of the object b = ~b2 [4] "
           << "of type bool is true." << endl;
   else
      cout << "The value of the object b = ~b2 [4] "
           << "of type bool is false." << endl;

   // Example of b = x [i] storing the value of the bit at
   // position i of bitset x in the object b of type bool
```

```
   b = b2 [4];
   if ( b )
      cout << "The value of the object b = b2 [4] "
           << "of type bool is true." << endl;
   else
      cout << "The value of the object b = b2 [4] "
           << "of type bool is false." << endl;

   // Example of x [i] . flip ( ) toggling the value of the bit at
   // position i of bitset x
   cout << "Before flipping the value of the bit at position 4 in "
        << "bitset b2,\nit is ( "<<  b2  << " )." << endl;
   b2 [4].flip( );
   cout << "After flipping the value of the bit at position 4 in "
        << "bitset b2,\nit becomes ( "<<  b2  << " )." << endl;
   bool c;
   c = b2 [4].flip( );
   cout << "After a second flip, the value of the position 4 "
        << "bit in b2 is now: " << c << ".";
}
```

Output

```
The initialized bitset<5> b1( 2 ) is: ( 00010 ).
The initialized bitset<5> b2( 6 ) is: ( 00110 ).
The bitset<5> b1 with the bit at position 0 set to 1 is: ( 00011 )
The bitset<5> b2 with the bit at position 4 set to the value
of the bit at position 0 of the bit in bitset<5> b1 is: ( 10110 )
The value of the object b = ~b2 [4] of type bool is false.
The value of the object b = b2 [4] of type bool is true.
Before flipping the value of the bit at position 4 in bitset b2,
it is ( 10110 ).
After flipping the value of the bit at position 4 in bitset b2,
it becomes ( 00110 ).
After a second flip, the value of the position 4 bit in b2 is now: 1.
```

`reset`

Resets all the bits in a `bitset` to 0 or resets a bit at a specified position to 0.

C++

```
bitset<N>& reset();
bitset<N>& reset(size_t pos);
```

Parameters

pos

The position of the bit in the `bitset` to be reset to 0.

Return Value

A copy of the `bitset` for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified is greater than the size of the `bitset`.

Example

C++

```
// bitset_reset.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 13 );
   cout << "The set of bits in bitset<5> b1(13) is: ( "<< b1 << " )"
        << endl;

   bitset<5> b1r3;
   b1r3 = b1.reset( 2 );
   cout << "The collecion of bits obtained from resetting the\n"
        << "third bit of bitset b1 is: ( "<< b1r3 << " )"
        << endl;

   bitset<5> b1r;
   b1r = b1.reset( );
   cout << "The collecion of bits obtained from resetting all\n"
        << "the elements of the bitset b1 is: ( "<< b1r << " )"
        << endl;
}
```

Output

```
The set of bits in bitset<5> b1(13) is: ( 01101 )
The collecion of bits obtained from resetting the
third bit of bitset b1 is: ( 01001 )
```

```
The collecion of bits obtained from resetting all
the elements of the bitset b1 is: ( 00000 )
```

set

Sets all the bits in a `bitset` to 1 or sets a bit at a specified position to 1.

C++

```
bitset<N>& set();

bitset<N>& set(
    size_t pos,
    bool val = true);
```

Parameters

pos

The position of the bit in the `bitset` to be set the assigned value.

val

The value to be assigned to the bit at the specified position.

Return Value

A copy of the `bitset` for which the member function was invoked.

Remarks

The second member function throws an out_of_range exception if the position specified is greater than the size of the `bitset`.

Example

C++

```
// bitset_set.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
```

```
   using namespace std;

   bitset<5> b1 ( 6 );
   cout << "The set of bits in bitset<5> b1(6) is: ( "<< b1 << " )"
        << endl;

   bitset<5> b1s0;
   b1s0 = b1.set( 0 );
   cout << "The collecion of bits obtained from setting the\n"
        << "zeroth bit of bitset b1 is: ( "<< b1s0 << " )"
        << endl;

   bitset<5> bs1;
   bs1 = b1.set( );
   cout << "The collecion of bits obtained from setting all the\n"
        << "elements of the bitset b1 is: ( "<< bs1 << " )"
        << endl;
}
```

Output

```
The set of bits in bitset<5> b1(6) is: ( 00110 )
The collecion of bits obtained from setting the
zeroth bit of bitset b1 is: ( 00111 )
The collecion of bits obtained from setting all the
elements of the bitset b1 is: ( 11111 )
```

`size`

Returns the number of bits in a `bitset` object.

C++

```
size_t size() const;
```

Return Value

The number of bits, `N`, in a `bitset<N>`.

Example

C++

```
// bitset_size.cpp
// compile with: /EHsc
#include <bitset>
```

```
#include <iostream>

int main()
{
    using namespace std;

    bitset<5> b1(6);
    size_t i;

    cout << "The set of bits in bitset<5> b1( 6 ) is: ( "<< b1 << " )"
         << endl;

    i = b1.size();

    cout << "The number of bits in bitset b1 is: " << i << "."
         << endl;
}
```

Output

```
The set of bits in bitset<5> b1( 6 ) is: ( 00110 )
The number of bits in bitset b1 is: 5.
```

`test`

Tests whether the bit at a specified position in a `bitset` is set to 1.

C++

```
bool test(size_t pos) const;
```

Parameters

pos

The position of the bit in the `bitset` to be tested for its value.

Return Value

`true` if the bit specified by the argument position is set to 1; otherwise, **`false`**.

Remarks

The member function throws an out_of_range

`to_string`

Converts a `bitset` object to a string representation.

C++

```
template <class charT = char, class traits = char_traits<charT>, class
Allocator = allocator<charT> >
   basic_string<charT, traits, Allocator> to_string(charT zero = charT('0'),
charT one = charT('1')) const;
```

Return value

A string object of class `basic_string`, where each bit set in the `bitset` has a corresponding character of 1, and a character of 0 if the bit is unset.

Example

C++

```
// bitset_to_string.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The ordered set of bits in the bitset<5> b1( 7 )"
        << "\n  that was generated by the number 7 is: ( "
        << b1 << " )" << endl;

   string s1;
   s1 =  b1.template to_string<char,
   char_traits<char>, allocator<char> >( );
   cout << "The string returned from the bitset b1"
        << "\n  by the member function to_string( ) is: "
        << s1 << "." << endl;
}
```

Output

```
The ordered set of bits in the bitset<5> b1( 7 )
  that was generated by the number 7 is: ( 00111 )
The string returned from the bitset b1
  by the member function to_string( ) is: 00111.
```

`to_ullong`

Returns an **`unsigned long long`** value that contains the same bits set as the contents of the `bitset` object.

C++

```
unsigned long long to_ullong() const;
```

Return value

Returns the sum of the bit values that are in the bit sequence as an **`unsigned long long`**. This **`unsigned long long`** value would re-create the same set bits if it's used to initialize a `bitset`.

Exceptions

Throws an overflow_error object if any bit in the bit sequence has a bit value that can't be represented as a value of type **`unsigned long long`**.

Remarks

Returns the sum of the bit values that are in the bit sequence as an **`unsigned long long`**.

`to_ulong`

Converts a `bitset` object to the integer that would generate the sequence of bits contained if used to initialize the `bitset`.

C++

```
unsigned long to_ulong( ) const;
```

Return value

An integer that would generate the bits in a `bitset` if used in the initialization of the `bitset`.

Remarks

Applying the member function would return the integer that has the same sequence of 1 and 0 digits as is found in sequence of bits contained in the `bitset`.

The member function throws an overflow_error object if any bit in the bit sequence has a bit value that can't be represented as a value of type **`unsigned long`**.

Example

C++

```
// bitset_to_ulong.cpp
// compile with: /EHsc
#include <bitset>
#include <iostream>

int main( )
{
   using namespace std;

   bitset<5> b1 ( 7 );

   cout << "The ordered set of bits in the bitset<5> b1( 7 )"
        << "\n  that was generated by the number 7 is: ( "
        << b1 << " )" << endl;

   unsigned long int i;
   i = b1.to_ulong( );
   cout << "The integer returned from the bitset b1,"
        << "\n  by the member function to_long( ), that"
        << "\n  generated the bits as a base two number is: "
        << i << "." << endl;
}
```

Output

```
The ordered set of bits in the bitset<5> b1( 7 )
  that was generated by the number 7 is: ( 00111 )
The integer returned from the bitset b1,
  by the member function to_long( ), that
  generated the bits as a base two number is: 7.
```

<cassert>

Article • 12/07/2021

Includes the C Standard library header <assert.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C Standard library header are declared in the `std` namespace.

> **Note**
>
> <assert.h> doesn't define the `static_assert` macro.

Syntax

```
C++

#include <cassert>
```

Macros

```
C++

#define assert(E)
```

Remarks

`assert(E)` is only constant, if NDEBUG is defined where `assert` is last defined or redefined, or *E* converted to bool evaluates to `true`.

See also

assert Macro, _assert, _wassert
Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<ccomplex>

Article • 12/07/2021

Includes the C++ standard library header <complex>.

> **Note**
>
> The C standard library <complex.h> header isn't included by <ccomplex>, because it's effectively replaced by the C++ overloads in <complex> and <cmath>. That makes the <ccomplex> header redundant. The <complex.h> header is deprecated in C++. The <ccomplex> header is deprecated in C++17 and removed in the draft C++20 standard.

Requirements

Header: <ccomplex>

Namespace: std

Remarks

The name `clog`, which is declared in <complex.h>, isn't defined in the `std` namespace because of potential conflicts with the `clog` that's declared in <iostream>.

See also

<complex>
<cmath>
Header files reference
C++ standard library overview
Thread safety in the C++ standard library

<cctype>

Article • 12/07/2021

Includes the Standard C library header <ctype.h> and adds the associated names to the `std` namespace.

Requirements

Header: <cctype>

Namespace: std

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Functions

C++

```
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cerrno>

Article • 12/07/2021

Includes the C Standard library header <errno.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C Standard library header are declared in the `std` namespace.

Syntax

C++

```
#include <cerrno>
```

Macros

C++

```
#define errno
#define E2BIG
#define EACCES
#define EADDRINUSE
#define EADDRNOTAVAIL
#define EAFNOSUPPORT
#define EAGAIN
#define EALREADY
#define EBADF
#define EBADMSG
#define EBUSY
#define ECANCELED
#define ECHILD
#define ECONNABORTED
#define ECONNREFUSED
#define ECONNRESET
#define EDEADLK
#define EDESTADDRREQ
#define EDOM
#define EEXIST
#define EFAULT
#define EFBIG
#define EHOSTUNREACH
#define EIDRM
#define EILSEQ
#define EINPROGRESS
#define EINTR
#define EINVAL
#define EIO
```

```
#define EISCONN
#define EISDIR
#define ELOOP
#define EMFILE
#define EMLINK
#define EMSGSIZE
#define ENAMETOOLONG
#define ENETDOWN
#define ENETRESET
#define ENETUNREACH
#define ENFILE
#define ENOBUFS
#define ENODATA
#define ENODEV
#define ENOENT
#define ENOEXEC
#define ENOLCK
#define ENOLINK
#define ENOMEM
#define ENOMSG
#define ENOPROTOOPT
#define ENOSPC
#define ENOSR
#define ENOSTR
#define ENOSYS
#define ENOTCONN
#define ENOTDIR
#define ENOTEMPTY
#define ENOTRECOVERABLE
#define ENOTSOCK
#define ENOTSUP
#define ENOTTY
#define ENXIO
#define EOPNOTSUPP
#define EOVERFLOW
#define EOWNERDEAD
#define EPERM
#define EPIPE
#define EPROTO
#define EPROTONOSUPPORT
#define EPROTOTYPE
#define ERANGE
#define EROFS
#define ESPIPE
#define ESRCH
#define ETIME
#define ETIMEDOUT
#define ETXTBSY
#define EWOULDBLOCK
#define EXDEV
```

Remarks

The macros here are defined by the POSIX standard.

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<cfenv>

Article • 12/07/2021

Includes the Standard C library header <fenv.h> and adds the associated names to the `std` namespace.

Syntax

```
C++

#include <cfenv>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants and Types

```
C++

#define FE_ALL_EXCEPT see below
#define FE_DIVBYZERO see below
#define FE_INEXACT see below
#define FE_INVALID see below
#define FE_OVERFLOW see below
#define FE_UNDERFLOW see below
#define FE_DOWNWARD see below
#define FE_TONEAREST see below
#define FE_TOWARDZERO see below
#define FE_UPWARD see below
#define FE_DFL_ENV see below

namespace std {
    using fenv_t = object type ;
    using fexcept_t = integer type ;
}
```

Functions

```
C++
```

```
int feclearexcept(int except);
int fegetexceptflag(fexcept_t* pflag, int except);
int feraiseexcept(int except);
int fesetexceptflag(const fexcept_t* pflag, int except);
int fetestexcept(int except);
int fegetround();
int fesetround(int mode);
int fegetenv(fenv_t* penv);
int feholdexcept(fenv_t* penv);
int fesetenv(const fenv_t* penv);
int feupdateenv(const fenv_t* penv);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cfloat>

Article • 12/07/2021

Includes the Standard C library header <float.h>.

Syntax

C++

```
#include <cfloat>
```

Definitions

C++

```
#define FLT_ROUNDS
#define FLT_EVAL_METHOD
#define FLT_HAS_SUBNORM
#define DBL_HAS_SUBNORM
#define LDBL_HAS_SUBNORM
#define FLT_RADIX
#define FLT_MANT_DIG
#define DBL_MANT_DIG
#define LDBL_MANT_DIG
#define FLT_DECIMAL_DIG
#define DBL_DECIMAL_DIG
#define LDBL_DECIMAL_DIG
#define DECIMAL_DIG
#define FLT_DIG
#define DBL_DIG
#define LDBL_DIG
#define FLT_MIN_EXP
#define DBL_MIN_EXP
#define LDBL_MIN_EXP
#define FLT_MIN_10_EXP
#define DBL_MIN_10_EXP
#define LDBL_MIN_10_EXP
#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP
#define FLT_MAX_10_EXP
#define DBL_MAX_10_EXP
#define LDBL_MAX_10_EXP
#define FLT_MAX
#define DBL_MAX
#define LDBL_MAX
#define FLT_EPSILON
```

```
#define DBL_EPSILON
#define LDBL_EPSILON
#define FLT_MIN
#define DBL_MIN
#define LDBL_MIN
#define FLT_TRUE_MIN
#define DBL_TRUE_MIN
#define LDBL_TRUE_MIN
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<charconv>

Article • 12/07/2021

Quickly convert a character sequence to an integer or floating-point value, and the other way around. One way to use this library is to write and roundtrip floating-point values in JSON and text files.

The conversion functions are tuned for performance, and also support shortest-round-trip behavior. Shortest-round-trip behavior means when a number is converted to chars, only enough precision is written out to enable recovering the original number when converting those chars back to a floating-point. No other CRT or STL function provides this capability.

Some of the benefits of using the `<charconv>` library are:

- The sequence of chars representing a numeric value doesn't need to be null-terminated. Likewise, when a number is converted to chars, the result isn't null-terminated.
- Conversion functions don't allocate memory. You own the buffer in all cases.
- Conversion functions don't throw. They return a structure that contains error information.
- Conversions aren't runtime rounding-mode sensitive.
- Conversions aren't locale aware. They always print and parse decimal points as '.' never as ',' for locales that use commas.

Requirements

Header: <charconv>

Namespace: std

/std:c++17 or later is required.

Members

Types

Type	Description
chars_format	Specifies formatting type such as scientific, hex, and so on.

Type	Description
from_chars_result	Holds the result of a `from_chars` conversion.
to_chars_result	Holds the result of a `to_chars` conversion.

Functions

Function	Description
from_chars	Convert chars to an integer, float, or double.
to_chars	Convert an integer, float, or double to chars.

See also

Header Files Reference

`<charconv>` functions

Article • 12/07/2021

The <charconv> header includes the following non-member functions:

Non-member functions	Description
to_chars	Convert an integer or floating-point value to a sequence of `char`.
from_chars	Convert a sequence of `char` to an integer or floating-point value.

These conversion functions are tuned for performance, and also support shortest-round-trip behavior. Shortest-round-trip behavior means when a number is converted to chars, only enough precision is written out to enable recovering the original number when converting those chars back to a floating-point.

- When converting chars to a number, the numeric value doesn't need to be null-terminated. Likewise, when converting a number to chars, the result isn't null-terminated.
- The conversion functions don't allocate memory. You own the buffer in all cases.
- The conversion functions don't throw. A result is returned from which you can determine if the conversion succeeded.
- The conversion functions aren't runtime rounding-mode sensitive.
- The conversion functions aren't locale aware. They always print and parse decimal points as `'.'`, and never as ',' for locales that use commas.

`to_chars`

Convert an integer or floating-point value to a sequence of `char`.

Converts `value` into a character string by filling the range [`first`, `last`), where [`first`, `last`) must be a valid range. Returns a to_chars_result structure. If the conversion is successful, as indicated by `to_char_result.ec`, the member `ptr` is the one-past-the-end pointer of the characters written. Otherwise, `to_char_result.ec` has the value `errc::value_too_large`, `to_char_result.ptr` has the value `last`, and the contents of the range [`first`, `last`) are unspecified.

The only way that `to_chars` can fail is if you provide an insufficiently large buffer to hold the result.

```
C++
```

```
// integer to chars

to_chars_result to_chars(char* first, char* last, char value, int base =
10);
to_chars_result to_chars(char* first, char* last, signed char value, int
base = 10);
to_chars_result to_chars(char* first, char* last, unsigned char value, int
base = 10);
to_chars_result to_chars(char* first, char* last, short value, int base =
10);
to_chars_result to_chars(char* first, char* last, unsigned short value, int
base = 10);
to_chars_result to_chars(char* first, char* last, int value, int base = 10);
to_chars_result to_chars(char* first, char* last, unsigned int value, int
base = 10);
to_chars_result to_chars(char* first, char* last, long value, int base =
10);
to_chars_result to_chars(char* first, char* last, unsigned long value, int
base = 10);
to_chars_result to_chars(char* first, char* last, long long value, int base
= 10);
to_chars_result to_chars(char* first, char* last, unsigned long long value,
int base = 10);
to_chars_result to_chars(char* first, char* last, bool value, int base = 10)
= delete;

// floating-point to chars

to_chars_result to_chars(char* first, char* last, float value);
to_chars_result to_chars(char* first, char* last, double value);
to_chars_result to_chars(char* first, char* last, long double value);
to_chars_result to_chars(char* first, char* last, float value, chars_format
fmt);
to_chars_result to_chars(char* first, char* last, double value, chars_format
fmt);
to_chars_result to_chars(char* first, char* last, long double value,
chars_format fmt);
to_chars_result to_chars(char* first, char* last, float value, chars_format
fmt, int precision);
to_chars_result to_chars(char* first, char* last, double value, chars_format
fmt, int precision);
to_chars_result to_chars(char* first, char* last, long double value,
chars_format fmt, int precision);
```

Parameters

first

Points to the beginning of the buffer to fill.

last

Points one char past the end of the buffer to fill.

value
The value to convert. If `value` is negative, the representation starts with `-`.

base
For integer conversions, the base to use when converting `value` to chars. Must be between 2 and 36, inclusive. There will be no leading zeros. Digits in the range 10..35 (inclusive) are represented as lowercase characters a..z

fmt
For floating-point conversions, a bitmask specifying the conversion format to use such as scientific, fixed, or hexadecimal. See chars_format for details.

precision
For floating-point conversions, the number of digits of precision for the converted value.

Return value

A to_chars_result containing the result of the conversion.

Remarks

Functions taking a chars_format parameter determine the conversion specifier as if they were using `printf()` as follows: The conversion specifier is `'f'` if `fmt` is `chars_format::fixed`, `'e'` if `fmt` is `chars_format::scientific`, `'a'` (without the leading `0x` in the result) if `fmt` is `chars_format::hex`, and `'g'` if `fmt` is `chars_format::general`. Specifying the shortest fixed notation may still result in lengthy output because it may be the shortest possible representation when the value is very large or very small.

The following table describes the conversion behavior given different combinations of `fmt` and `precision` parameters. The term "shortest-round-trip behavior" refers to writing the fewest number of digits necessary such that parsing that representation using the corresponding `from_chars` function will recover the value exactly.

`fmt` and `precision` combination	**Output**
Neither	Whichever of fixed or scientific notation is shorter, preferring fixed as a tiebreaker. This behavior can't be simulated by any overload that takes the `fmt` parameter.
`fmt`	The shortest-round-trip behavior for the specified format, such as the shortest scientific format.

`fmt` and `precision` combination	Output
`fmt` and `precision`	Uses the given precision, following `printf()` style, without shortest-round-trip behavior.

Example

C++

```
#include <charconv>
#include <stdio.h>
#include <system_error>

template <typename T> void TestToChars(const T t)
{
    static_assert(std::is_floating_point_v<T>);
    constexpr bool IsFloat = std::is_same_v<T, float>;

    char buf[100]; // 100 is large enough for double and long double values
because the longest possible outputs are "-1.23456735e-36" and
"-1.2345678901234567e-100".
    constexpr size_t size = IsFloat ? 15 : 24;
    const std::to_chars_result res = std::to_chars(buf, buf + size, t);  //
points to buffer area it can use. Must be char, not wchar_t, etc.

    if (res.ec == std::errc{}) // no error
    {
        // %.*s provides the exact number of characters to output because
the output range, [buf, res.ptr), isn't null-terminated
        printf("success: %.*s\n", static_cast<int>(res.ptr - buf), buf);
    }
    else // probably std::errc::value_too_large
    {
        printf("Error: %d\n", static_cast<int>(res.ec));
    }
}

int main()
{
    TestToChars(123.34);
    return 0;
}
```

`from_chars`

Convert a sequence of `char` to an integer or floating-point value.

C++

```
// char to an integer value

from_chars_result from_chars(const char* first, const char* last, char&
value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, signed
char& value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, unsigned
char& value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, short&
value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, unsigned
short& value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, int&
value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, unsigned
int& value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, long&
value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, unsigned
long& value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, long long&
value, int base = 10);
from_chars_result from_chars(const char* first, const char* last, unsigned
long long& value, int base = 10);

// char to a floating-point value

from_chars_result from_chars(const char* first, const char* last, float&
value, chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double&
value, chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long
double& value, chars_format fmt = chars_format::general);
```

Parameters

first

Points to the beginning of the buffer of characters to convert.

last

Points one past the end element of the buffer of characters to convert.

value

If the conversion is successful, contains the result of the conversion.

base

For integer conversions, the base to use during the conversion. Must be between 2 and 36, inclusive.

fmt
For floating-point conversions, the format of the sequence of chars being converted. See chars_format for details.

Remarks

The `from_chars()` functions analyze the string [`first`, `last`) for a number pattern, where [`first`, `last`) is required to be a valid range.

When parsing chars, whitespace isn't ignored. Unlike `strtod()`, for example, the buffer must start with a valid numeric representation.

Returns a from_chars_result structure.

If no characters match a number pattern, `value` is unmodified, `from_chars_result.ptr` points to `first`, and `from_chars_result.ec` is `errc::invalid_argument`.

If only some characters match a number pattern, `from_chars_result.ptr` points to the first character not matching the pattern, or has the value of the `last` parameter if all characters match.

If the parsed value isn't in the range representable by the type of `value`, `value` is unmodified and `from_chars_result.ec` is `errc::result_out_of_range`.

Otherwise, `value` is set to the parsed value, after rounding, and `from_chars_result.ec` is equal to `errc{}`.

Example

C++

```
#include <charconv>
#include <stdio.h>
#include <string_view>
#include <system_error>

double TestFromChars(const std::string_view sv)
{
    const char* const first = sv.data();
    const char* const last = first + sv.size();
    double dbl;

    const std::from_chars_result res = std::from_chars(first, last, dbl);

    if (res.ec == std::errc{}) // no error
    {
```

```
        printf("success: %g\n", dbl);
    }
    else
    {
        printf("Error: %d\n", static_cast<int>(res.ec));
    }

    return dbl;
}

int main()
{
    double dbl = TestFromChars("123.34");
    return 0;
}
```

Requirements

Header: <charconv>

Namespace: std

/std:c++17 or later is required.

See also

<charconv>
The shortest decimal string that round-trips printf() format specifiers

chars_format enum

Article • 08/17/2021

Used with the <charconv> library to specify the floating-point format for primitive numerical conversions.

Syntax

C++

```
enum class chars_format {
    scientific = unspecified,
    fixed = unspecified,
    hex = unspecified,
    general = fixed | scientific
};
```

Members

Element	Description
`scientific`	Causes `from_chars()` to expect and parse an exponent. It is like the `printf()` format specifier `'e'`, which formats for scientific notation, like `"1.729e+01"`.
`fixed`	Causes `from_chars()` not to expect or parse an exponent. It is like the `printf()` format specifier `'f'`, which formats for floating-point, like `"17.29"`.
`hex`	Causes `from_chars()` to expect the number in hexadecimal format, but without a leading `0x`.
`general`	Causes `from_chars()` to accept (but not require) an exponent. For `to_chars()`, it is like the `printf()` format specifier `'g'`, which switches between scientific notation or fixed. It takes into consideration what the exponent will be so that it can generate reasonably compact output. For example: `1e-5` results in `"1e-05"`, but `1e-4` results in `"0.001"`. `1e5` results in `100000`, while `1e6` results in `1e+06`. `1e0` produces `1`.

Remarks

For the from_chars functions, this enum describes what kind of input to expect. For the to_chars functions, it describes what kind of output to emit.

Requirements

Header: <charconv>

Namespace: std

/std:c++17 or later is required.

See also

<charconv>
printf() format specifiers

from_chars_result struct

Article • 08/17/2021

Syntax

C++

```
struct from_chars_result {
    const char* ptr;
    errc ec;
};
```

Member	Description
`ptr`	If `ec` is equal to `errc{}`, the conversion was successful and `ptr` points to the first character that isn't part of the recognized number.
`ec`	The conversion error code. For specific error codes, see errc.

Remarks

An example: parsing `"1729cats"` as a decimal integer will succeed, and `ptr` will point to `'c'` which is the first non-digit and is also one-past-the-end of `"1729"`.

If no characters match a number pattern, `from_chars_result.ptr` points to `first`, and `from_chars_result.ec` is `errc::invalid_argument`.

If only some characters match a number pattern, `from_chars_result.ptr` points to the first character not matching the pattern, or has the value of the `last` parameter if all characters match.

If the parsed value won't fit the range for the type of conversion being done, `from_chars_result.ec` is `errc::result_out_of_range`.

Requirements

Header: <charconv>

Namespace: std

Compiler option: /std:c++17 or later is required.

See also

from_chars

to_chars_result struct

Article • 08/17/2021

Syntax

C++

```
struct to_chars_result {
    char* ptr;
    errc ec;
};
```

Members

Member	Description
`ptr`	If `ec` is equal to `errc{}`, the conversion was successful and `ptr` is the one-past-the-end pointer of the written characters. Otherwise, `ptr` has the value of the `to_chars()` parameter `last`, and the contents of the range [first, last) are unspecified.
`ec`	The conversion error code. For specific error codes, see errc.

Requirements

Header: <charconv>

Namespace: std

Compiler option: /std:c++17 or later is required.

See also

to_chars

<chrono>

Article • 06/06/2022

Include the standard header `<chrono>` to define classes and functions that represent and manipulate time durations and time instants.

Beginning in Visual Studio 2015, the implementation of `steady_clock` has changed to meet the C++ Standard requirements for steadiness and monotonicity:

- `steady_clock` is now based on `QueryPerformanceCounter()`
- `high_resolution_clock` is now a typedef for `steady_clock` In the Microsoft C++ implementation, `steady_clock::time_point` is now a `typedef` for `chrono::time_point<steady_clock>`. However, this isn't necessarily the case for other implementations.

Requirements

Header: `<chrono>`

Regional legislatures occasionally make changes to time zones, and an international standards body specifies when new leap seconds should be accounted for. A database of these changes was added to Windows 10. Specifically:

Feature	Client version	Server version
Leap second database updates	Windows 10 version 1809 or later	Windows Server 2019 or later
Time zone support	Windows 10 version 1903/19H1 or later	Windows Server 2022 or later

Using the time-zone facilities on older versions of Windows results in a runtime error.

Namespace: `std`

Calendrical types

Name	Description
common_type struct	Describes specializations of class template common_type for instantiations of `duration` and `time_point`.

Name	Description
day class	A day of the month. For example, the 25th day of the month.
duration class	A time interval.
duration_values struct	Provides specific values for the `duration` template parameter `Rep`.
hh_mm_ss class	Splits a duration into hours:minutes:seconds.
last_spec	Used to indicate the last item in a month such as last day of the week of a month (the last Tuesday of February 2020) or the last day of a month (the last day of April 2019).
leap_second class	A date and a value for an inserted leap second.
leap_second_info struct	The data returned by get_leap_second_info.
month class	A month of a year. For example, July.
month_day class	A specific day of a specific month. For example, July 30.
month_day_last class	The last day of a month.
month_weekday class	The nth weekday of a specific month.
month_weekday_last class	The nth weekday of a specific month.
time_point class	A point in time.
weekday class	A day of the week.
weekday_last class	The last weekday of a month.
weekday_indexed class	Combines a day of the week with an index that represents the weekday of the month.
year class	A year in the Gregorian calendar.
year_month class	A year and month. The day isn't specified.
year_month_day class	A year, month, and day.
year_month_day_last class	The last day of a specific month and year.
year_month_weekday class	A specific year, month, and nth weekday of the month.
year_month_weekday_last class	A specific year, month, and last weekday of the month.

Clocks

Name	Description
file_clock class	An alias for the clock used for `std::filesystem::file_time_type`, which is used to express file timestamps.
gps_clock class	A clock that keeps GPS time. Measures time starting from the first Sunday of January 1980 at 00:00:00 UTC.
high_resolution_clock struct	A clock with a nanosecond tick period.
local_t struct	A pseudo-clock used as an argument to the `time_point` template to indicate that the `time_point` represents local time.
steady_clock struct	A `steady` clock. This clock is preferred for measuring time intervals.
system_clock struct	A clock based on the real-time clock of the system.
tai_clock class	Measures International Atomic Time (TAI) starting from Thursday, January 1, 1958 at 00:00:00. This clock doesn't account for leap seconds.
utc_clock class	Measures time since 00:00:00 UTC on Thursday, January 1, 1970. This clock accounts for leap seconds, and is the time standard used around the world.

Time zones

Name	Description
choose	An enum that specifies how to resolve the ambiguous conversion of a `local_time` to a `sys_time`.
local_info	Provides a low-level interface to time zone information about the result of converting a `local_time` to a `sys_time`.
sys_info	Provides a low-level interface to time zone information about the result of converting a `sys_time` to a `local_time`.
time_zone class	All time zone transitions for a specific geographic area.
time_zone_link class	An alternative name for a `time_zone`.
tzdb struct	Represents a copy of the time zone database.
tzdb_list class	A singleton list of time zone databases.

Name	Description
zoned_time class	A pairing of a time_zone and a time_point with a specified precision.
zoned_traits struct	Used to associate a different default time zone with a `zoned_time`, and optionally map a custom name to that default time zone.

Exceptions

Name	Description
ambiguous_local_time	Error thrown when a `local_time` is converted to a `sys_time` and the result is ambiguous.
nonexistent_local_time	Error thrown when a `local_time` is converted to a `sys_time` and the result is a time that doesn't exist.

Functions

Name	Description
ceil(duration)	Returns the `ceil` of a `duration` object as a specified type.
ceil(time_point)	Returns the `ceil` of a `time_point` object as a specified type.
clock_cast	Converts a time_point from one clock to an equivalent `time_point` for another clock.
duration_cast	Casts a `duration` object to a specified type.
floor(duration)	Returns the `floor` of a `duration` object as a specified type.
floor(time_point)	Returns the `floor` of a `time_point` object as a specified type.
from_stream()	Parse the input stream into one of the `std::chrono` time or interval types such as `day`, `month`, `weekday`, `year`, and so on, using the specified format.
get_leap_second_info	Gets a leap_second_info struct.
is_am	Whether an `hours` object is AM.
is_pm	Whether an `hours` object is PM.
locate_zone	Gets a specified time zone.
make12	Returns an `hours` in 12-hour form.
make24	Returns an `hours` in 24-hour form.

Name	Description
round(duration)	Returns a `duration` object rounded as a specified type.
round(time_point)	Returns a `time_point` object rounded as a specified type.
time_point_cast	Casts a `time_point` object to a specified type.

Time zone related

Name	Description
current_zone	Gets the current time zone.
get_tzdb	Gets the first time zone database entry.
get_tzdb_list	Gets the list of time zone database entries.
reload_tzdb	Gets an updated time zone database entry.
remote_version	Checks for an updated time zone database entry.

Operators

Name	Description
operator-	Subtract or negate various `<chrono>` objects.
operator!=	Inequality operator that is used with various `<chrono>` objects.
operator modulo	Operator for modulo operations on `duration` objects.
operator*	Multiplication operator for `duration` objects.
operator/	Division operator for `duration` objects.
operator/	Provides syntax to create calendar dates.
operator+	Add to various `<chrono>` objects.
operator<	Determines whether various `<chrono>` objects are less than another.
operator<=	Determines whether various `<chrono>` objects are less than or equal to another.
operator==	Determines whether various `<chrono>` objects are equal to each other.
operator>	Determines whether various `<chrono>` objects are greater than another.

Name	Description
operator>=	Determines whether various `<chrono>` objects are greater than or equal to another.

Typedefs

For more information about ratio types that are used in the following typedefs, see <ratio>.

Convenience `duration` types

Name	Description
`typedef duration<long long, nano> nanoseconds;`	Synonym for a `duration` type that has a tick period of one billionth (1/1,000,000,000) of a second.
`typedef duration<long long, micro> microseconds;`	Synonym for a `duration` type that has a tick period of one-millionth (1/1,000,000) of a second.
`typedef duration<long long, milli> milliseconds;`	Synonym for a `duration` type that has a tick period of one-thousandth (1/1,000) of a second.
`typedef duration<long long> seconds;`	Synonym for a `duration` type that has a tick period of 1 second.
`typedef duration<int, ratio<60>> minutes;`	Synonym for a `duration` type that has a tick period of 1 minute.
`typedef duration<int, ratio<3600>> hours;`	Synonym for a `duration` type that has a tick period of 1 hour.

Convenience `time_point` types

Name	Description
`file_time`	A synonym for `time_point<file_clock, Duration>`. Represents a time_point for a file_clock. You specify the `Duration`, for example, `file_time<seconds> ft;`.
`gps_seconds`	A synonym for `gps_time<seconds>;` A count of seconds, represented by a `time_point` that is associated with a gps_clock.
`gps_time`	A synonym for `time_point<gps_clock, Duration>`. Represents a `time_point` for a gps_clock. You specify the `Duration`, for example, `gps_time<milliseconds> gps;`.
`local_days`	A synonym for `local_time<days>`. A count of days, represented by a time_point that isn't associated with any time zone.

Name	Description
`local_seconds`	A synonym for `local_time<seconds>`.
`local_time`	A synonym for `time_point<local_t, Duration>`. Represents a `time_point` for a local time that isn't associated with a time zone yet. You specify the `Duration`, for example, `local_time<seconds> lt;`. A `local_time` is a local time *somewhere*. It isn't the current local time of your computer's clock. Only when you pair a `local_time` with a `time_zone` do you get a point in time that can be converted to UTC time, or the time in a specific time zone.
`sys_days`	A synonym for `sys_time<days>`. A count of days since the system_clock's epoch, represented by a `time_point` that is associated with a system_clock.
`sys_seconds`	A synonym for `sys_time<seconds>`. A count of non-leap seconds since the epoch of system_clock (Jan 1, 1970 00:00:00 UTC), represented by a `time_point` that is associated with a `system_clock`.
`sys_time`	A synonym for `time_point<system_clock, Duration>`. You specify the `Duration`, for example, `sys_time<seconds> st;`. Represents a `time_point` returned from system_clock::now(). It represents Unix time, which closely approximates UTC time.
`tai_seconds`	A synonym for `tai_time<seconds>`. A count of seconds, represented by a `time_point` that is associated with a tai_clock.
`tai_time`	A synonym for `time_point<tai_clock, Duration>`. You provide the `Duration`, for example, `tai_time<seconds> tt;`. Represents a `time_point` for a tai_clock.
`utc_seconds`	A synonym for `utc_time<seconds>;`
`utc_time`	A synonym for `time_point<utc_clock, Duration>`. You provide the `Duration`, for example, `utc_time<seconds> ut;`. Represents a `time_point`for a utc_clock.

Type traits

Name	Description
clock_time_conversion	A trait that specifies how to convert a time_point from one clock to another.
is_clock	Check if a type is a clock.
treat_as_floating_point	Check if a `duration` can be converted to another `duration` that has a different tick period.

Literals

(C++11) The `<chrono>` header defines the following user-defined literals that you can use for greater convenience, type-safety, and maintainability of your code. These literals are defined in the `literals::chrono_literals` inline namespace and are in scope when `std::chrono` is in scope.

Declaration	Description
`hours operator "" h(unsigned long long Val)`	Specifies hours as an integral value.
`duration<double, ratio<3600>> operator "" h(long double Val)`	Specifies hours as a floating-point value.
`minutes (operator "" min)(unsigned long long Val)`	Specifies minutes as an integral value.
`duration<double, ratio<60>> (operator "" min)( long double Val)`	Specifies minutes as a floating-point value.
`seconds operator "" s(unsigned long long Val)`	Specifies minutes as an integral value.
`duration<double> operator "" s(long double Val)`	Specifies seconds as a floating-point value.
`milliseconds operator "" ms(unsigned long long Val)`	Specifies milliseconds as an integral value.
`duration<double, milli> operator "" ms(long double Val)`	Specifies milliseconds as a floating-point value.
`microseconds operator "" us(unsigned long long Val)`	Specifies microseconds as an integral value.
`duration<double, micro> operator "" us(long double Val)`	Specifies microseconds as a floating-point value.
`nanoseconds operator "" ns(unsigned long long Val)`	Specifies nanoseconds as an integral value.
`duration<double, nano> operator "" ns(long double Val)`	Specifies nanoseconds as a floating-point value.

The following examples show how to use `<chrono>` literals:

C++

```
constexpr auto day = 24h;
constexpr auto week = 24h* 7;
constexpr auto my_duration_unit = 108ms;
```

See also

Header Files Reference

`<chrono>` functions

Article • 10/17/2022

`abs(duration)`

Returns `d` if `d >= d.zero()`; otherwise returns `-d`.

Syntax

```
C++

template <class Rep, class Period>
constexpr duration<Rep, Period> abs(duration<Rep, Period> d ); // C++17
```

Parameters

`Rep`

The type of the internal representation of the source `duration` *`d`*.

`Period`

A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

`d`

The source `duration` object.

Return value

The absolute value of `d`.

Example: `abs(duration)`

```
C++

// compile using: /std:c++latest
#include <chrono>
#include <iostream>

int main()
{
```

```
    std::cout << abs(-24h);
    return 0;
}
```

Output

```
24h
```

`ceil(duration)`

Returns the smallest representable `duration` in the target type that's greater than or equal to the specified `duration`.

Syntax

C++

```
template<class ToDuration, class Rep, class Period>
constexpr ToDuration
ceil(const duration<Rep, Period>& d);  // C++17
```

Parameters

`ToDuration`

The target `duration` type. Constrained as a specialization of `duration`.

`Rep`

The type of the internal representation of the source `duration` *`d`*.

`Period`

A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

`d`

The source `duration` object.

Return value

Returns the smallest `duration` representable in `ToDuration` that's greater than or equal to the parameter *`d`*.

Remarks

`ceil` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration.

`ceil(time_point)`

Returns the smallest time point representable in the target `duration` that's greater than or equal to the specified time point.

Syntax

C++

```
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
ceil(const time_point<Clock, Duration>& t);  // C++17
```

Parameters

`ToDuration`
The target `duration` type. Constrained as a specialization of `duration`.

`Clock`
The common clock type of the result and the source parameter *`tp`*.

`Duration`
The `duration` type of *`tp`*.

`tp`
The source `time_point` object.

Return value

Returns the smallest time point representable using `ToDuration` that's greater than or equal to *`tp`*. Effectively, `time_point<Clock, ToDuration>(ceil<ToDuration>(tp.time_since_epoch()));`.

Remarks

`ceil` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration.

`clock_cast`

Converts a time_point for one clock to an equivalent `time_point` for another clock.

Syntax

```
C++

template <class DestClock, class SourceClock, class Duration>
auto clock_cast(const time_point<SourceClock, Duration>& t); // C++ 20
```

Parameters

`DestClock`
The clock type to convert the `time_point` to.

`Duration`
The duration of the `SourceClock`, or one that you specify.

`SourceClock`
The clock type that the `time_point` to convert is based on.

`t`
The `time_point` to convert.

Return value

A time_point equivalent to **`t`**, but specific to `DestClock`.

Remarks

The parameters `SourceClock` and `Duration` can be inferred via class template argument deduction when not explicitly passed. For example, given `clock_cast<utc_clock>(file_clock::now())`, `SourceClock` is deduced to be `file_clock`, and `Duration` is deduced to be `file_clock::duration`.

From the following list of well-formed clock conversions, the one that requires the fewest conversion steps to get from the `SourceClock` to the `DestClock` is selected.

C++

```
clock_time_conversion<DestClock, SourceClock>{}(t)

clock_time_conversion<DestClock, system_clock>{}(
    clock_time_conversion<system_clock, SourceClock>{}(t))

clock_time_conversion<DestClock, utc_clock>{}(
    clock_time_conversion<utc_clock, SourceClock>{}(t))

clock_time_conversion<DestClock, utc_clock>{}(
    clock_time_conversion<utc_clock, system_clock>{}(
        clock_time_conversion<system_clock, SourceClock>{}(t)))

clock_time_conversion<DestClock, system_clock>{}(
    clock_time_conversion<system_clock, utc_clock>{}(
        clock_time_conversion<utc_clock, SourceClock>{}(t)))
```

For more information about what `clock_time_conversion` does, see clock_time_conversion struct.

Example `clock_cast`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    utc_clock::time_point t = clock_cast<utc_clock>(file_clock::now());
    std::cout << t;

    return 0;
}
```

Output

```
2021-10-11 22:58:17.8540720
```

`current_zone`

Gets the current time zone object.

Syntax

```
C++
```

```
const time_zone* current_zone();  // C++20
```

Return value

Returns a pointer to a time_zone as if by a call to `get_tzdb().current_zone()`. It throws a `runtime_error` exception if it's the first reference to the time zone database and the time zone database can't be initialized.

`duration_cast`

Casts a duration to the specified target `duration` type.

Syntax

```
C++
```

```
template <class ToDuration, class Rep, class Period>
constexpr ToDuration
duration_cast(const duration<Rep, Period>& d);  // C++11
```

Parameters

`ToDuration`

The target `duration` type. Constrained as a specialization of `duration`.

`Rep`

The type of the internal representation of the source `duration` *`d`*.

`Period`

A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

d

The source `duration` object to cast to the target `duration` type.

Return value

An object of type `ToDuration` that represents the `duration` *d*. It's truncated if necessary to fit into the target type. The result of casting a floating-point `duration` to an integral `duration` is undefined if the source contains a *NaN*, an infinity, or is too large for representation in the target `duration`.

Remarks

You don't need to use `duration_cast` to convert between `duration` types when the source period is exactly divisible by the target period, such as when you convert minutes to seconds. Also, you don't need it to convert between floating-point `duration` types. You can do both conversions by using ordinary casts or the duration constructors.

`duration_cast` doesn't participate in overload resolution unless `ToDuration` is an instance of duration. It does all conversions by using **`static_cast`** instead of implicit conversions. Multiplications and divisions are avoided if possible. For example, when the compiler knows that the common ratio of the target and source periods has a numerator or denominator of 1. Computations are done in the widest type available, then converted as if by **`static_cast`** to the result type when finished.

Example `duration_cast`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    seconds s(1);
    std::cout << duration_cast<microseconds>(s) << '\n';
    std::cout << duration_cast<nanoseconds>(s) << '\n';

    return 0;
}
```

Output

```
1000000us
1000000000ns
```

`floor(duration)`

Returns the greatest representable `duration` in the target type that's less than or equal to the specified `duration`.

Syntax

C++

```
template<class ToDuration, class Rep, class Period>
constexpr ToDuration
floor(const duration<Rep, Period>& d);  // C++17
```

Parameters

`ToDuration`

The target `duration` type. Constrained as a specialization of `duration`.

`Rep`

The type of the internal representation of the source `duration` *`d`*.

`Period`

A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

`d`

The source `duration` object.

Return value

Returns the greatest `duration` representable in `ToDuration` that's less than or equal to the parameter *`d`*.

Remarks

`floor` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration.

`floor(time_point)`

Returns the largest time point representable in the target `duration` that's less than or equal to the specified time point.

Syntax

```
C++
```

```
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
floor(const time_point<Clock, Duration>& tp);  // C++17
```

Parameters

`ToDuration`
The target `duration` type. Constrained as a specialization of `duration`.

`Clock`
The common clock type of the result and the source parameter *`tp`*.

`Duration`
The `duration` type of *`tp`*.

`tp`
The source `time_point` object.

Return value

Returns the largest time point representable using `ToDuration` that's less than or equal to *`tp`*. Effectively, `time_point<Clock, ToDuration>(floor<ToDuration>(tp.time_since_epoch()));`.

Remarks

`floor` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration.

`from_stream`

Parse the input stream into one of the `std::chrono` time or interval types such as `day`, `month`, `month_day`, `weekday`, `year`, `year_month`, `year_month_day`, and so on, using the specified format.

If the parse fails, `is.setstate(ios_base::failbit)` is called and the output parameter isn't modified.

C++

```
// 1) day - C++20
template<class charT class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    day& d, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 2) duration - C++20
template<class charT, class traits, class Rep, class Period, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    duration<Rep, Period>& dur, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 3) file_time - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    file_time<Duration>& ft, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 4) gps_time - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    gps_time<Duration>& gt, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 5) local_time - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
```

```
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    local_time<Duration>& lt, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 6) month - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    month& m, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 7) month_day - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    month_day& md, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 8) utc_time  - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    utc_time<Duration>& ut, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 9) sys_time - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    sys_time<Duration>& st, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 10) tai_time - C++20
template<class charT, class traits, class Duration, class Alloc =
allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    tai_time<Duration>& tt, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);

// 11) weekday - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
```

```
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    weekday& wd, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 12) year - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    year& y, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 13) year_month - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    year_month& ym, basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 14) year_month_day - C++20
template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(
    basic_istream<charT, traits>& is, const charT* fmt,
    year_month_day& ymd, basic_string<charT, traits, Alloc>* abbrev =
nullptr,
    minutes* offset = nullptr);
```

Template parameters

`Alloc`

The type that represents the allocator object that handles the string's allocation and deallocation of memory.

`charT`

The data type of a single character to be read from the stream and stored in the string. The C++ Standard Library provides specializations of this class template, with the type definitions string for elements of type `char`, wstring, for `wchar_t`, u16string for `char16_t`, and u32string for `char32_t`.

`traits`

Describes `charT` attributes for the `basic_string` and `basic_istream` specialization.

`Rep`

The internal representation type of a `duration` type.

Period

A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

Duration

The `duration` type used for time specialization.

Parameters

abbrev

If *abbrev* isn't `nullptr`, and the format specifier `%Z` is specified, and the parse is successful, then *abbrev* contains the parsed value.

d

If the parse is successful, contains the parsed day when the function returns.

dur

The `duration` parsed from the stream.

fmt

The format string used to match the input. See Parse format strings for the list of parse formatting options.

ft

The `file_time` parsed from the stream.

gt

The gps_time parsed from the stream.

is

The input stream to parse.

lt

The local_time parsed from the stream.

m

The `month` parsed from the stream.

md

The `month_day` parsed from the stream.

offset

If *offset* isn't `nullptr`, and the format specifier `%z` or modified variant such as `%Ez` or

`%0z` is specified, and the parse is successful, then *`offset`* points to the parsed value.

`st`

The sys_time parsed from the stream.

`tt`

The tai_time parsed from the stream.

`ut`

The utc_time parsed from the stream.

`wd`

The `weekday` parsed from the stream.

`y`

The year parsed from the stream.

`ym`

The `year_month` parsed from the stream.

`ymd`

The `year_month_day` parsed from the stream.

Return value

The input stream, *`is`*

Example: `from_stream`

C++

```
// compile using: /std:c++latest
#include <chrono>
#include <iostream>

int main()
{
    std::istringstream str{ "22" };
    std::basic_istream<char> stream{ str.rdbuf() };
    std::chrono::day d;
    std::chrono::from_stream(stream, "%d", d);
    std::cout << d << "\n";
    return 0;
}
```

Output

22

Remarks

7) If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

12) If `%Z` is used and successfully parsed, that value will be assigned to `*abbrev` if `abbrev` is non-null. If `%z` (or a modified variant) is used and successfully parsed, that value will be assigned to `*offset` if `offset` is non-null.

`from_stream` format strings

The format may be one of these strings:

Date

Specifier	Description
`%D`	Equivalent to `%m/%d/%y`
`%F` `%`*N*`F`	Equivalent to `%Y-%m-%d`. If modified with a width `N`, the width is applied to only `%Y`.
`%x` `%Ex`	The locale's date representation. `%Ex` parses the locale's alternate date representation.[1]

Day

Specifier	Description
`%d` `%Od` `%`*N*`d` `%e` `%Oe` `%`*N*`e`	The day of the month as a decimal number. `%`*N*`d` specifies the maximum number of characters to read, for example `%1d`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%Od` (letter `O`, not zero) interprets the locale's alternative representation of the day of the month.[1] `%e` is equivalent to `%d` and can be modified like `%d`.[1]

Day of the week

Specifier	Description
`%a` `%A`	The locale's full or abbreviated case-insensitive weekday name. `%A` is equivalent to `%a`
`%u` `%N u`	The ISO weekday as a decimal number (1-7), where Monday is 1. `%N u` specifies the maximum number of characters to read, for example `%2u`. If *N* isn't specified, the default is 1. Leading zeros are permitted but not required.
`%w` `%N w` `%Ow`	The weekday as a decimal number (0-6), where Sunday is 0. `%N w` specifies the maximum number of characters to read, for example `%2w`. If *N* isn't specified, the default is 1. Leading zeroes are permitted but not required. `%Ow` (letter `O`, not zero) interprets the locale's alternative representation.[1]

Week/day of the year

Specifier	Description
`%j` `%N j`	If the type being formatted is a specialization of duration, the decimal number of days without padding. Otherwise, the day of the year as a decimal number. `Jan 1` is `001`. If the result is fewer than three digits, it's left-padded with `0` (zero) to three digits. `%N j`specifies the maximum number of characters to read, for example `%2j`. If *N* isn't specified, the default is 3. Leading digits are allowed but aren't required.
`%U` `%N U` `%OU`	The week number of the year as a decimal number. The first Sunday of the year is the first day of week `01`. Days of the same year before that week are in week `00`. If the result is a single digit, it's prefixed with `0` (zero). `%N U` specifies the maximum number of characters to read, for example `%2U`. If *N* isn't specified, the default is 2. Leading zeros are permitted but no required. `%OU` (letter `O`, not zero) parses the locale's alternative representation.[1]
`%W` `%N W` `%OW`	The week number of the year as a decimal number. The first Monday of the year is the first day of week `01`. Days of the same year before that week are in week `00`. If the result is a single digit, it's prefixed with `0` (zero). `%N W` specifies the maximum number of characters to read, for example `%2W`. If *N* isn't specified, the default is 1 Leading zeros are permitted but no required.`%OW` (letter `O`, not zero) parses the locale's alternative representation.[1]

Time of day

Specifier	Description

Specifier	Description
`%H` `%`*N*`H` `%OH`	The hour (24-hour clock) as a decimal number. If the result is a single digit, it's prefixed with a `0` (zero). `%`*N*`H` specifies the maximum number of characters to read, for example, `%1H`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%OH` (letter `O`, not zero) parses the locale's alternative representation.[1]
`%I` `%`*N*`I` `%OI`	The hour (12-hour clock) as a decimal number. If the result is a single digit, it's prefixed with `0` (zero). `%`*N*`I` specifies the maximum number of characters to read, for example, `%1I`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%OI` (letter `O`, not zero) parses the locale's alternative representation.[1]
`%M` `%`*N*`M` `%OM`	The minutes as a decimal number. If the result is a single digit, it's prefixed with `0` (zero). `%`*N*`M` specifies the maximum number of characters to read, for example `%3M`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%OM` (letter `O`, not zero) parses the locale's alternative representation.[1]
`%S` `%`*N*`S` `%OS`	Seconds as a decimal number. If the number of seconds is less than 10, the result is prefixed with `0` (zero). If the precision of the input can't be exactly represented with seconds, then the format is a decimal floating-point number with a fixed format. It has a microseconds precision if the function can't convert the floating-point decimal seconds within 18 fractional digits. Otherwise, its precision matches the precision of the input. The character for the decimal point is localized according to the locale. `%`*N*`S` specifies the maximum number of characters to read, for example `%3S`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%OS` (letter `O`, not zero) parses the locale's alternative representation.[1]
`%p`	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
`%r`	The locale's 12-hour clock time.
`%R`	Equivalent to `%H:%M`.
`%T`	Equivalent to `"%H:%M:%S"`.
`%X`, `%EX`	The locale's time representation. `%EX` parses the alternate locale's time representation.[1]

Month

Specifier	Description

Specifier	Description
`%b`, `%B`, `%h`	The locale's full or abbreviated month name. If the value doesn't contain a valid month, a `format_error` exception is thrown. `%h` is equivalent to `%b`.
`%m`, `%`*N*`m`, `%Om`	The month as a decimal number. Jan is 1. `%`*N*`m` specifies the maximum number of characters to read, for example, `%3m`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%Om` (letter `O`, not zero) interprets the locale's alternative representation.[1]

Year

Specifier	Description
`%C`, `%`*N*`C`, `%EC`	The century as a decimal number. `%`*N*`C` specifies the maximum number of characters to read, for example, `%1N`. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%EC` interprets the locale's alternative representation of the century.
`%y`, `%`*N*`y`, `%Ey`, `%Oy`	The last two decimal digits of the year. If the century isn't otherwise specified (for example, by using `%C`), values in the range `[69, 99]` are presumed to refer to the years 1969 to 1999, and values in the range `[00, 68]` are presumed to refer to the years 2000 to 2068. `%`*N*`y` specifies the maximum number of characters to read. If *N* isn't specified, the default is 2. Leading zeroes are permitted but not required. `%Ey` and `%Oy` (letter `O`, not zero) interpret the locale's alternative representation.[1]
`%Y`, `%`*N*`Y`, `%EY`,	The year as a decimal number. If the result is fewer than four digits, it's left-padded with `0` (zero) to four digits. `%`*N*`Y` specifies the maximum number of characters to read. If *N* isn't specified, the default is 4. `%EY` parses the locale's alternative full year representation.[1]

ISO 8601 week-based year

In ISO 8601, weeks begin with Monday. The first week of the year must include January 4 and include the first Thursday of the year.

Specifier	Replacement
`%g` `%`*N*`g`	The last two decimal digits of the ISO week-based year. If the result is a single digit, is prefixed by `0` (zero). `%`*N*`g` specifies the maximum number of characters to read, for example, `%1g`. If *N* isn't specified, the default is 2

Specifier	Replacement
`%G` `%NG`	The ISO week-based year as a decimal number. If the result is fewer than four digits, it's left-padded with `0` (zero) to four digits. `%NG` specifies the maximum number of characters to read, for example, `%1G`. If *N* isn't specified, the default is 4
`%V` `%OV` `%NV`	The ISO week-based week number as a decimal number. If the result is a single digit, it's prefixed with `0` (zero). `%NV` specifies the maximum number of characters to read, for example, `%1V`. If *N* isn't specified, the default is 2 `%OV` (letter `O`, not zero) parses the locale's alternative representation.[1]

General

Specifier	Replacement
`%%`	Matches the % character
`%c` `%Ec`	The locale's date and time representation. `%Ec` interprets the locale's alternate date and time representation.[1]
`%n`	Matches a new-line character
`%t`	Matches zero or one whitespace character
`%z` `%Ez` `%Oz`	The offset from UTC in the format `[+|-]hh[mm]`. For example, `-0430` refers to 4 hours 30 minutes behind UTC, and `04` refers to 4 hours ahead of UTC. `%Ez` and `%Oz` (letter `O`, not zero) parse a `:` between the hours and minutes and render leading zeroes on the hour field optional[1]: `[+|-]h[h][:mm]`. For example, `-04:30` refers to 4 hours 30 minutes behind UTC, and 4 refers to 4 hours ahead of UTC.
`%Z`	The time zone abbreviation or name. A single word is parsed. This word can only contain alphanumeric characters from the basic source character set, or one of `_`, `/`, `-`, or `+`.

Flags by type

Class	Specifier/Flag
day	d, e
duration	j, H, I, M, S, r, R, T, p, (q, Q are only for format, not parsing)
`file_time`	Z, z, c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p

Class	Specifier/Flag
`gps_time`	Z, z, c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p
hh_mm_ss	H, I, M, S, r, R, T, p
`local_time`	c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p
`local_time_format_t`	z, Z, c, x, X, D, F, Y, C, y, b, B, h, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p, g, G, U, V, W
`local_info`	z, Z
month	b, h, B, m
month_day	B, d, j, e, b, h, m
month_day_last	B, d, j, e, b, h, m
month_weekday	b, B, h, m, a, A, u, w
month_weekday_last	b, B, h, m, a, A, u, w
`sys_info`	z, Z
`sys_time`	Z, z, c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p
`tai`	Z, z, c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p
`utc_time`	Z, z, c, x, X, D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p
weekday	a, A, u, w
weekday_indexed	a, A, u, w
weekday_last	a, A, u, w
year	Y, y, C
`year_month`	Y, y, B, g, G, h, C, b, m
`year_month_day`	D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w
`year_month_day_last`	D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w
`year_month_weekday`	D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w
`year_month_weekday_last`	D, F, g, G, j, U, V, W, Y, y, C, b, h, B, m, d, e, a, A, u, w

Class	Specifier/Flag
`zoned_time`	z, Z, c, x, X, D, F, Y, C, y, b, B, h, m, d, e, a, A, u, w, H, I, M, S, r, R, T, p, g, G, U, V, W

[1]The implementation is currently consistent with strftime in that although `O` (letter O) and `e` are accepted, they're ignored. That is, `"%Od"` is interpreted as `"%d"`.

`get_leap_second_info`

Returns a leap_second_info for the specified time. This provides info about whether the supplied time occurs during a leap second insertion. It also provides the number of leap seconds that have been added between January 1, 1970 and the specified time. A leap second insertion occurs when the International Earth Rotation and Reference Systems Service (IERS) declares that a leap second will be added (resulting in a 61 second minute) to account for the difference between atomic time and time tracked by measuring the rotation of the earth, which is irregular, and is gradually slowing.

Syntax

C++

```
template<class Duration>
leap_second_info
get_leap_second_info(const utc_time<Duration>& ut);  // C++20
```

Parameters

`ut`
The source `utc_time` to get the `leap_second_info` for.

Return value

Returns a `leap_second_info` whose member `is_leap_second` is **`true`** if *`ut`* is during a positive leap second insertion; otherwise, **`false`**. The `elapsed` member holds the sum of leap seconds between the epoch date `1970-01-01` and *`ut`*. If `is_leap_second` is **`true`**, the leap second referred to by *`ut`* is included in the `elapsed` sum.

`get_tzdb`

Gets the first entry in the program-wide time zone database list.

Syntax

```
C++
```

```
const tzdb& get_tzdb();  // C++20
```

Return value

Returns a reference to the first `tzdb` object as if by a call to `get_tzdb_list().front()`. It throws a `runtime_error` exception if it's the first reference to the time zone database and the time zone database can't be initialized.

Remarks

If it hasn't been initialized yet, `get_tzdb` initializes the program-wide time zone database on first access. Upon initialization, the database is a `tzdb_list` that holds a single initialized `tzdb` object.

`get_tzdb_list`

Gets the program-wide singleton time zone database.

Syntax

```
C++
```

```
tzdb_list& get_tzdb_list();  // C++20
```

Return value

Returns a reference to the program-wide `tzdb_list` object. It throws a `runtime_error` exception if it can't return a reference to a valid `tzdb_list` object.

Remarks

If it hasn't been initialized yet, `get_tzdb_list` initializes the program-wide time zone database on first access. Upon initialization, the database is a `tzdb_list` that holds a single initialized `tzdb` object. The `get_tzdb_list` function is thread-safe.

`is_am`

Predicate for whether the specified hour is in the *ante-meridiem* (AM) part of the day.

Syntax

```
C++
constexpr bool is_am(const hours& h) noexcept;  // C++20
```

Parameters

h

The source `hours` representation in 24-hour time notation.

Return value

Returns **`true`** if `0h <= h && h <= 11h`, otherwise **`false`**.

`is_pm`

Predicate for whether the specified hour is in the *post-meridiem* (PM) part of the day.

Syntax

```
C++
constexpr bool is_pm(const hours& h) noexcept;  // C++20
```

Parameters

h

The source `hours` representation in 24-hour time notation.

Return value

Returns `true` if `12h <= h && h <= 23h`, otherwise `false`.

`locate_zone`

Gets a time zone object specified by time zone name.

Syntax

```
C++
```

```
const time_zone* locate_zone(string_view tz_name);  // C++20
```

Parameters

`tz_name`
The name of the time zone to return.

Return value

Returns a pointer to a time_zone as if by a call to `get_tzdb().locate_zone(tz_name)`. It throws a `runtime_error` exception if it can't find the specified time zone, or if it's the first reference to the time zone database and the time zone database can't be initialized.

Remarks

If it hasn't been initialized yet, `locate_zone` initializes the program-wide time zone database on first access. Upon initialization, the database is a `tzdb_list` that holds a single initialized `tzdb` object.

`make12`

Returns the specified hour in 12-hour time notation.

Syntax

```
C++
```

```
constexpr hours make12(const hours& h) noexcept;  // C++20
```

Parameters

h

The source `hours` representation in 24-hour time notation.

Return value

Returns the 12-hour equivalent of *h* in the range `[1h, 12h]`. The return value is unspecified if *h* isn't in the range `[0h, 23h]`.

make24

Returns the specified hour in 24-hour time notation.

Syntax

C++

```
constexpr hours make24(const hours& h, bool is_pm) noexcept;  // C++20
```

Parameters

h

The source `hours` representation in 12-hour time notation.

is_pm

The source `hours` representation is PM (if **`true`**) or AM (if **`false`**).

Return value

If `is_pm` is **`false`**, `make24` returns the 24-hour equivalent of *h* in the range `[0h, 11h]`, assuming *h* represents an AM hour. Otherwise, it returns the 24-hour equivalent of *h* in the range `[12h, 23h]`, assuming *h* represents a PM hour. The return value is unspecified if *h* isn't in the range `[1h, 12h]`.

`reload_tzdb`

Reloads the time zone database if updated information is available.

Syntax

```
C++
```

```
const tzdb& reload_tzdb();
```

Return value

After it makes a time zone database update, if any, `reload_tzdb` returns a reference to the first `tzdb` object as if by a call to `get_tzdb_list().front()`. It throws a `runtime_error` exception if it can't return a reference to a valid `tzdb` object.

Remarks

The *local* time zone database is the one supplied when the program first accesses the database, such as by a call to `current_zone`. While the program is running, the implementation may update the time zone database. The update doesn't affect the program in any way unless the program calls the `reload_tzdb` function. The potentially updated time zone database is called the *remote* time zone database.

The `reload_tzdb` function checks the version of both the local time zone database and the remote time zone database. If the versions of the local and remote databases are the same, it makes no changes. Otherwise, the remote database is pushed to the front of the `tzdb_list` accessed by `get_tzdb_list`. An update doesn't invalidate any pointers, references, or iterators. The `reload_tzdb` function is thread-safe for calls to `get_tzdb_list().front()` and `get_tzdb_list().erase_after()`.

`remote_version`

Gets a string that contains the latest remote database version.

Syntax

```
C++
```

```
string remote_version();
```

Return value

Returns a `string` that contains the latest remote database version.

`round(duration)`

Rounds the specified `duration` to the nearest representable `duration` in the target type.

Syntax

C++

```
template<class ToDuration, class Rep, class Period>
constexpr ToDuration
round(const duration<Rep, Period>& d);  // C++17
```

Parameters

`ToDuration`
The target `duration` type. Constrained as a specialization of `duration` where `treat_as_floating_point_v<typename ToDuration::rep>` is **`false`**.

`Rep`
The type of the internal representation of the source `duration` *`d`*.

`Period`
A std::ratio type representing the ratio of one second to the source `Rep` type (that is, seconds per `Rep`).

`d`
The source `duration` object.

Return value

Returns the nearest `duration` representable in `ToDuration` to the parameter *`d`*. Ties go to the even value, that is, the value `t` where `t % 2 == 0`.

Remarks

`round` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration, and `ToDuration` has an integral representation.

`round(time_point)`

Returns the nearest time point representable in the target `duration` to the specified time point.

```
C++
```

```
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
round(const time_point<Clock, Duration>& tp);
```

Parameters

`ToDuration`
The target `duration` type. Constrained as a specialization of `duration` where `treat_as_floating_point_v<typename ToDuration::rep>` is **`false`**.

`Clock`
The common clock type of the result and the source parameter *`tp`*.

`Duration`
The `duration` type of *`tp`*.

`tp`
The source `time_point` object.

Return value

Returns the nearest time point representable using `ToDuration` to *`tp`*. Ties go to the even value, that is, the value `t` where `t % 2 == 0`. Effectively, `time_point<Clock, ToDuration>(round<ToDuration>(tp.time_since_epoch()));`.

Remarks

`round` doesn't participate in overload resolution unless the `ToDuration` type is an instance of a duration.

`time_point_cast`

Casts a time_point object to a `time_point` that has a different `duration` type.

Syntax

```
C++
```

```
template <class ToDuration, class Clock, class Duration>
time_point<Clock, ToDuration>
time_point_cast(const time_point<Clock, Duration>& tp);  // C++11
```

Parameters

`ToDuration`
The target `duration` type. Constrained as a specialization of `duration`.

`Clock`
The common clock type of the result and the source parameter *`tp`*.

`Duration`
The `duration` type of *`tp`*.

`tp`
The `time_point` object to cast to one that has the `ToDuration` type.

Return value

Returns a `time_point` object that has a `ToDuration` type. Effectively, `time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()));`.

Remarks

Unless `ToDuration` is a specialization of duration, this function doesn't participate in overload resolution.

See also

<chrono>
chrono literals
chrono operators
duration class
time_point class
time_zone class

`<chrono>` operators

Article • 12/06/2021

`operator+`

Addition operator for the following types:

- day
- duration
- month
- time_point
- weekday
- year
- year_month
- year_month_day
- year_month_day_last
- year_month_weekday
- year_month_weekday_last

C++

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2,
Period2>>::type
   operator+(
      const duration<Rep1, Period1>& Left,
      const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, typename common_type<Duration1, duration<Rep2,
Period2>>::type>
   operator+(
      const time_point<Clock, Duration1>& Time,
      const duration<Rep2, Period2>& Dur);

3)
template <class Rep1, class Period1, class Clock, class Duration2>
time_point<Clock, constexpr typename common_type<duration<Rep1, Period1>,
Duration2>::type>
   operator+(
      const duration<Rep1, Period1>& Dur,
      const time_point<Clock, Duration2>& Time);

4)
```

```
constexpr day operator+(const day& d, const days& ds) noexcept; // C++ 20
constexpr day operator+(const days& ds, const day&  d) noexcept; // C++ 20

5)
constexpr month operator+(const month& m, const months& ms) noexcept; // C++
20
constexpr month operator+(const months& ms, const month& m) noexcept; // C++
20

6)
constexpr weekday operator+(const weekday& wd, const days& wds) noexcept //
C++ 20
constexpr weekday operator+(const days& ds, const weekday& wd) noexcept; //
C++ 20

7)
constexpr year operator+(const year& y, const years& ys) noexcept; // C++ 20
constexpr year operator+(const years& ys, const year& y) noexcept; // C++ 20

8)
constexpr year_month operator+(const year_month& ym, const months& dm)
noexcept; // C++ 20
constexpr year_month operator+(const months& dm, const year_month& ym)
noexcept; // C++ 20
constexpr year_month operator+(const year_month& ym, const years& dy)
noexcept; // C++ 20
constexpr year_month operator+(const years& dy, const year_month& ym)
noexcept; // C++ 20

9)
constexpr year_month_day operator+(const year_month_day& ymd, const months&
dm) noexcept; // C++ 20
constexpr year_month_day operator+(const months& dm, const year_month_day&
ymd) noexcept; // C++ 20
constexpr year_month_day operator+(const year_month_day& ymd, const years&
dy) noexcept; // C++ 20
constexpr year_month_day operator+(const years& dy, const year_month_day&
ymd) noexcept; // C++ 20

10)
constexpr year_month_day_last operator+(const year_month_day_last& ymdl,
const months& dm) noexcept; // C++ 20

11)
constexpr year_month_day_last operator+(const months& dm, const
year_month_day_last& ymdl) noexcept; // C++ 20

12)
constexpr year_month_day_last operator+(const year_month_day_last& ymdl,
const years& dy) noexcept; // C++ 20
constexpr year_month_day_last operator+(const years& dy, const
year_month_day_last& ymdl) noexcept; // C++ 20

13)
constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const
```

```
months& dm) noexcept; // C++ 20
constexpr year_month_weekday operator+(const months& dm, const
year_month_weekday& ymwd) noexcept; // C++ 20

14)
constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const
years& dy) noexcept; // C++ 20

15)
constexpr year_month_weekday operator+(const years& dy, const
year_month_weekday& ymwd) noexcept; // C++ 20

16)
constexpr year_month_weekday_last operator+(const year_month_weekday_last&
ymwdl, const months& dm) noexcept; // C++ 20
constexpr year_month_weekday_last operator+(const months& dm, const
year_month_weekday_last& ymwdl) noexcept; // C++ 20

17)
constexpr year_month_weekday_last operator+(const year_month_weekday_last&
ymwdl, const years& dy) noexcept; // C++ 20
constexpr year_month_weekday_last operator+(const years& dy, const
year_month_weekday_last& ymwdl) noexcept; // C++ 20
```

Return value

1) After converting *Left* and *Right* to their common type, returns a `duration` with a tick count equal to the sum of the converted tick counts.

2-3) Return a `time_point` object that represents a point in time that is displaced by the interval *Dur* from the point in time *Time*.

4) Returns the result of `d+ds.count()`. If the result is out of the range [0, 255], then the result is unspecified.

5) Returns the result of `m+ms.count()`. If the result is out of the range [1, 12], it's reduced modulo 12 and then +1.

6) Returns the result of adding the number of days and weekdays to the `weekday`. The result will be modulo 7, so always in the range [0,6]

7) Returns the result of adding the year to the specified number of years.

8) Returns the result of adding the number of months and years to the specified month and year.

9) Returns the result of adding months or years to a `year_month_day`. If `ymd.month()` is `February` and `ymd.day()` is not in the range [1d, 28d], `ok()` may return `false` for the

result of the addition.

10) Returns `(ymdl.year() / ymdl.month() + dm) / last`. Note: The `/` used here isn't a division operator. It's the date operator.

11) Returns `ymdl + dm`.

12) Returns `{ymdl.year()+dy, ymdl.month_day_last()}`

13) Returns `ymwd + dm.count()`.

14-15) Returns `{ymwd.year()+dy, ymwd.month(), ymwd.weekday_indexed()}`.

16) Returns `(ymwdl.year() / ymwdl.month() + dm) / ymwdl.weekday_last()`. Note: The `/` used here isn't a division operator but the date operator.

17) Returns: `ymwdl + dy`

Example: `operator+`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    // day
    day d{1};
    std::cout << d + days(2) << '\n'; // 03

    // month
    month m{11};
    std::cout << m + months(3)<< '\n'; // Feb

    // weekday
    weekday wd = Thursday;
    std::cout << wd + days(1) << '\n'; // Fri

    // year_month_day_last
    year_month_day_last ymdl{June / last / 2021};
    std::cout << ymdl + years{1} + months{1} << '\n'; // 2022/Jul/last

    // year_month_weekday
    year_month_weekday ymw{ year(1997) / January / Wednesday[1] };
    std::cout << ymw + months{1} << '\n'; // 1997/Feb/Wed[1]
    std::cout << ymw + years{1} << '\n'; // 1998/Jan/Wed[1]
```

```
    // year_month_weekday_last
    year_month_weekday_last ymwl{ year(1997) / January / Wednesday[last] };
    std::cout << ymwl + months{ 1 } << '\n'; // 1997/Feb/Wed[last]
    std::cout << ymwl + years{ 1 } << '\n'; // 1998/Jan/Wed[last]

    return 0;
}
```

Output

```
03
Feb
Fri
2022/Jul/last
1997/Feb/Wed[1]
1998/Jan/Wed[1]
1997/Feb/Wed[last]
1998/Jan/Wed[last]
```

Unary `operator+`

Apply unary plus to the following types:

- duration.

C++

```
// duration
constexpr common_type_t<duration> operator+() const // C++ 20
```

Return value

Returns `*this`

`operator-`

Subtraction operator for the following types:

- day
- duration
- month
- time_point
- weekday

- year
- year_month
- year_month_day
- year_month_day_last
- year_month_weekday
- year_month_weekday_last

C++

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, Period1>, duration<Rep2,
Period2>>::type
   operator-(
       const duration<Rep1, Period1>& Left,
       const duration<Rep2, Period2>& Right);
2)
template <class Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, typename common_type<Duration1, duration<Rep2,
Period2>>::type
   operator-(
       const time_point<Clock, Duration1>& Time,
       const duration<Rep2, Period2>& Dur);
3)
template <class Clock, class Duration1, class Duration2>
constexpr typename common_type<Duration1, Duration2>::type
   operator-(
       const time_point<Clock, Duration1>& Left,
       const time_point<Clock, Duration2>& Right);
4)
constexpr day operator-(const day& d,  days& ds) noexcept; // C++ 20
constexpr day operator-(const day& d, const day& d) noexcept; // C++ 20

5)
constexpr month operator-(const month& m, const months& ms) noexcept; // C++
20
constexpr month operator-(const month& m, const month& ms) noexcept; // C++
20

6)
constexpr months operator-(const year_month& Left, const year_month& Right)
noexcept; // C++ 20

7)
constexpr weekday operator-(const weekday& Left, const days& Right)
noexcept; // C++ 20

8)
constexpr days operator-(const weekday& Left, const weekday& Right)
noexcept; // C++ 20

9)
```

```
constexpr year operator-(const year& y, const years& ys) noexcept; // C++ 20

10)
constexpr years operator-(const year& y, const year& y2) noexcept; // C++ 20

11)
constexpr year_month operator-(const year_month& ym, const months& dm)
noexcept; // C++ 20
constexpr year_month operator-(const year_month& ym, const years& dy)
noexcept; // C++ 20

12)
constexpr year_month_day operator-(const year_month_day& ymd, const months&
dm) noexcept; // C++ 20
constexpr year_month_day operator-(const year_month_day& ymd, const years&
dy) noexcept; // C++ 20

13)
constexpr year_month_day_last operator-(const year_month_day_last& ymdl,
const months& dm) noexcept;  // C++ 20

14)
constexpr year_month_day_last operator-(const year_month_day_last& ymdl,
const years& dy) noexcept;  // C++ 20

15)
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const
months& dm) noexcept; // C++ 20

16)
constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const
years& dy) noexcept; // C++ 20

17)
constexpr year_month_weekday_last operator-(const year_month_weekday_last&
ymwdl, const months& dm) noexcept; // C++ 20

18)
constexpr year_month_weekday_last operator-(const year_month_weekday_last&
ymwdl, const years& dy) noexcept; // C++ 20
```

Return value

1) After converting the durations being subtracted to their common type, returns a `duration` with a tick count equal to the number of ticks in *`Right`* subtracted from the number of ticks in *`Left`*.

2) Returns a `time_point` that represents a point in time that is displaced by the negation of the time interval that is represented by *`Dur`*, from the point in time that is specified by *`Time`*.

3) Returns a `duration` object that represents the time interval between *`Left`* and *`Right`*.

4) Returns the result of `d-ds.count()`. If the result is out of the range [0, 255], then the result is unspecified.

5) If `m.ok() == true` and `ms.ok() == true`, returns the result of subtracting the two month values, or subtracting the number of months. The result will be in the range [1, 12]. If the result is negative, it wraps around. For example, subtracting one month from January (`month m1{1} - months{1};` results in 12 (December).

6) Returns the difference in months between *`Left`* and *`Right`*

7) If `Left.ok() == true` and `Right.ok() == true`, returns a `weekday` in the range [`days{0}`, `days{6}`].

8) Returns the number of days between two weekdays.

9) Returns `year(int(y)-ys.count())`

10) Returns `years(int(y) - int(y2))`. Subtracting two `year` values results in a `std::chrono::years`, which represents the difference in years between `y` and `y2`. For example, `2021y-2000y` produces `years(21)`.

11) Returns the result of subtracting months or years from a `year_month` value.

12) Returns the result of subtracting months years from a `year_month_day` value.

13) Returns the result of subtracting the number of months from the `year_month_day_last` value. Essentially: `ymdl-dm`.

14) Returns the result of subtracting the number of years from the `year_month_day_last` value. Essentially: `ymdl-dy`.

15) Returns the result of subtracting the number of months from the `year_month_weekday` value. Essentially: `ymwd-dm`.

16) Returns the result of subtracting the number of years from the `year_month_weekday` value. Essentially: `ymwd-dy`.

17) Returns the result of subtracting the number of months from the `year_month_weekday_last` value. Essentially: `ymwdl-dm`.

18) Returns the result of subtracting the number of years from the `year_month_weekday_last` value. Essentially: `ymwdl-dy`.

Example: `operator-`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    // day
    day d{10};
    d = d - days(5);
    std::cout << d << '\n'; // 05

    // month
    month m{2};
    m = m - months{1};
    std::cout << m << '\n'; // Jan
    m = m - months{1};
    std::cout << m << '\n'; // Dec

    // year
    auto diff1 = 2021y-2000y;
    auto diff2 = 2021y-years{1};
    std::cout << diff1.count() << '\n'; // 21
    std::cout << diff2 << '\n'; // 2020

    // year_month
    const year theYear{ 2021 };
    year_month ym1{theYear, June};
    year_month ym2 = ym1 - months{2};
    std::cout << ym2 << '\n'; // 2021/Apr
    year_month ym3 = ym1 - years{2};
    std::cout << ym3 << '\n'; // 2019/Jun

    // year_month_day_last
    year_month_day_last ymdl = June / last / 2021;
    std::cout << ymdl - years{1} - months{1} << '\n'; // 2022/Jul/last

    // year_month_weekday
    year_month_weekday ymw{ year(1997) / January / Wednesday[1] };
    std::cout << ymw - months{1} << '\n'; // 1996/Dec/Wed[1]
    std::cout << ymw - years{1} << '\n'; // 1996/Jan/Wed[1]

    // year_month_weekday_last
    year_month_weekday_last ymwl{ year(1997) / January / Wednesday[last] };
    std::cout << ymwl - months{ 1 } << '\n'; // 1996/Dec/Wed[last]
    std::cout << ymwl - years{ 1 } << '\n'; // 1996/Jan/Wed[last]

    return 0;
}
```

Output

```
05
Jan
Dec
21
2020
2021/Apr
2019/Jun
2020/May/last
1996/Dec/Wed[1]
1996/Jan/Wed[1]
1996/Dec/Wed[last]
1996/Jan/Wed[last]
```

Unary `operator-`

Negates a duration.

C++

```
constexpr common_type_t<duration> operator-() const;
```

Return value

Returns a negated copy of `*this`

Example: unary `operator-`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
   duration<int, std::milli> milliseconds(120);
   std::cout << -milliseconds << '\n';
   return 0;
}
```

Output

```
-120ms
```

`operator!=`

Determines whether:

1) Two duration objects don't represent the same number of ticks.
2) Two time_point objects don't represent the same point in time.\

C++

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Duration2>
constexpr bool operator!=(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

`Left`
The left `duration` or `time_point` object.

`Right`
The right `duration` or `time_point` object.

Return value

1) Returns **`true`** if the number of ticks for the type common to *`Left`* and *`Right`* aren't equal. Otherwise, returns **`false`**.
2) Returns **`true`** if the two time_point objects don't represent the same point in time. Otherwise, returns **`false`**.\

`operator*`

Multiplication operator for duration objects. After converting the `duration`s being multiplied to their common type, returns a `duration` with a tick count equal to the multiplication of the converted tick counts.

C++

```
1)
template <class Rep1, class Period1, class Rep2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period1>
   operator*(
      const duration<Rep1, Period1>& Dur,
      const Rep2& Mult);

2)
template <class Rep1, class Rep2, class Period2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period2>
   operator*(
       const Rep1& Mult,
       const duration<Rep2,
       Period2>& Dur);
```

Parameters

Dur

A `duration` object.

Mult

An integral value.

Return value

Returns a `duration` object whose interval length is *Mult* multiplied by the length of *Dur*.

1) Unless `is_convertible<Rep2, common_type<Rep1, Rep2>>` *holds* *true*, this function doesn't participate in overload resolution. For more information, see <type_traits>.

2) Unless `is_convertible<Rep1, common_type<Rep1, Rep2>>` *holds* *true*, this function doesn't participate in overload resolution. For more information, see <type_traits>.

operator<

1) After converting the `duration`s being compared to their common type, determines if the number of ticks for `Left` is fewer than for `Right`.

2) Determines if the point in time since the epoch of the `Left` time_point is less than the time since the epoch of the `time_point` in `Right`.

```
C++
```

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Duration2>
constexpr bool operator<(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

`Left`
The left `duration` or `time_point` object.

`Right`
The right `duration` or `time_point` object.

Return value

1) Returns **`true`** if number of ticks for *`Left`* is less than the number of ticks for *`Right`*. Otherwise, the function returns **`false`**.

2) Returns **`true`** if *`Left`* precedes *`Right`*. Otherwise, returns **`false`**.

`operator<=`

1) After converting the `duration`s being compared to their common type, determines if the number of ticks for `Left` is fewer or the same as `Right`.

2) Determines if the point in time since the epoch of the `Left` time_point is less than or equal to the time since the epoch of the `time_point` in `Right`.

```
C++
```

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
```

```
constexpr bool operator<=(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Duration2>
constexpr bool operator<=(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

Left

The left `duration` or `time_point` object.

Right

The right `duration` or `time_point` object.

Return value

1) Returns **`true`** if number of ticks for *Left* is less than or equal to the number of ticks for *Right*. Otherwise, the function returns **`false`**.

2) Returns **`true`** if *Left* precedes, or is equal to, *Right*. Otherwise, returns **`false`**.

operator==

Determines whether:

1) duration objects represent time intervals that have the same length.
2) time_point objects represent the same point in time.
3) day objects represent the same day.
4) month objects represent the same month.
5) month_day objects represent the same month and day.
6) month_day_last objects represent the same month.
7) month_weekday objects represent the same month and nth weekday.
8) month_weekday_last objects represent the same month and last weekday.
9) weekday objects represent the same weekday.
10) weekday_last objects represent the same last weekday of the month.
11) weekday_indexed represent the same weekday index.
12) year represent the same year.
13) year_month represent the same year and month.

14) year_month_day represent the same year, month, and day.
15) year_month_day_last represent the same last day of the year and month.
16) year_month_weekday represent the same weekday, year, and month.
17) year_month_weekday_last represent the same last weekday of the month, year, and month.
18) time_zone_link have the same `name`. The `target` name isn't considered.
19) zoned_time represent the same time and time zone.

C++

```
// 1) duration<Rep, Period>
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

// 2) time_point
template <class Clock, class Duration1, class Duration2>
constexpr bool operator==(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);

// 3) day
constexpr bool operator==(const day& Left, const day& Right) noexcept; //
C++ 20

// 4) month
constexpr bool operator==(const month& Left, const month& Right) noexcept;
// C++ 20

// 5) month_day
constexpr bool operator==(const month_day& Left, const month_day& Right)
noexcept; // C++ 20

// 6) month_day_last
constexpr bool operator==(const month_day_last& Left, const month_day_last&
Right) noexcept; // C++ 20

// 7) month_weekday
constexpr bool operator==(const month_weekday& Left, const month_weekday&
Right) noexcept; // C++ 20

// 8) month_weekday_last
constexpr bool operator==(const month_weekday_last& Left, const
month_weekday_last& Right) noexcept; // C++ 20

// 9) weekday
constexpr bool operator==(const weekday& Left, const weekday& Right)
noexcept; // C++ 20

// 10) weekday_last
constexpr bool operator==(const weekday_last& Left, const weekday_last&
```

```
Right) noexcept; // C++ 20

// 11) weekday_indexed
constexpr bool operator==(const weekday_indexed& x, const weekday_indexed&
y) noexcept; // C++ 20

// 12) year
constexpr bool operator==(const year& Left, const year& y ) noexcept; // C++
20

// 13) year_month
constexpr bool operator==(const year_month& Left, const year_month& Right)
noexcept; // C++ 20

// 14) year_month_day
constexpr bool operator==(const year_month_day& Left, const year_month_day&
Right) noexcept; // C++ 20

// 15) year_month_day_last
constexpr bool operator==(const year_month_day_last& Left, const
year_month_day_last& Right) noexcept; // C++ 20

// 16) year_month_weekday
constexpr bool operator==(const year_month_weekday& Left, const
year_month_weekday& Right) noexcept; // C++ 20

// 17)  year_month_weekday_last
constexpr bool operator==(const year_month_weekday_last& Left, const
year_month_weekday_last& Right) noexcept; // C++ 20

// 18) time_zone_link
bool operator==(const time_zone_link& Left, const time_zone_link& Right)
noexcept;

// 19) zoned_time
template <class Duration1, class Duration2, class TimeZonePtr>
bool operator==(const zoned_time<Duration1, TimeZonePtr>& Left, const
zoned_time<Duration2, TimeZonePtr>& Right); // C++ 20
```

Parameters

Left

The left object to compare, for example, `Left == Right`

Right

The right object to compare.

Return value

1) Returns `true` if the number of ticks for the type common to *`Left`* and *`Right`* are equal. Otherwise, returns `false`.
2) Returns `true` if *`Left`* and *`Right`* represent the same point in time. Otherwise, returns `false`.
3-17) Returns `true` if *`Left`* and *`Right`* have the same value. Otherwise, returns `false`.
18) Returns `true` if `Left.name() == Right.name()`. Otherwise, returns `*false*`.
19) Returns `true` if `Left.get_time_zone() == _Right.get_time_zone() && Left.get_sys_time() == Right.get_sys_time();`

`operator>`

1) After converting the `duration`s being compared to their common type, determines if the number of ticks for `Left` is greater than for `Right`.

2) Determines if the point in time since the epoch of the `Left`time_point is greater than the time since the epoch of the `time_point` in `Right`.

```
C++

1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Duration2>
constexpr bool operator>(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

`Left`
The left `duration` or `time_point` object.

`Right`
The right `duration` or `time_point` object.

Return value

1) Returns `true` if number of ticks for *`Left`* is greater than the number of ticks for *`Right`*. Otherwise, the function returns `false`.

2) Returns `true` if *`Left`* comes after *`Right`*. Otherwise, returns `false`.

`operator>=`

1) After converting the `duration`s being compared to their common type, determines if the number of ticks for `Left` is greater than or equal to `Right`.

2) Determines if the point in time since the epoch of the `Left`time_point is greater than or equal to the time since the epoch of the `time_point` in `Right`.

C++

```
1)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(
    const duration<Rep1, Period1>& Left,
    const duration<Rep2, Period2>& Right);

2)
template <class Clock, class Duration1, class Duration2>
constexpr bool operator>=(
    const time_point<Clock, Duration1>& Left,
    const time_point<Clock, Duration2>& Right);
```

Parameters

`Left`
The left `duration` or `time_point` object.

`Right`
The right `duration` or `time_point` object.

Return value

1) Returns `true` if number of ticks for *`Left`* is greater than or equal to the number of ticks for *`Right`*. Otherwise, the function returns `false`.

2) Returns `true` if *`Left`* comes after, or is equal to, *`Right`*. Otherwise, returns `false`.

`operator<=>`

The spaceship operator, with `operator==`, synthesizes operators for `<`, `<=`, `>`, `>=`, and `!=` for the following types:

- day
- duration
- month
- month_day
- month_day_last
- time_point
- time_zone_link
- year
- year_month
- year_month_day_last

C++

```
1)
constexpr bool operator<=>(const day& Left, const day& Right) noexcept; // C++ 20

constexpr std::strong_ordering operator<=>(const month& Left, const month& Right) noexcept; // C++ 20

constexpr strong_ordering operator<=>(const month_day& Left, const month_day& Right) noexcept; // C++ 20

constexpr std::strong_ordering operator<=>(const year& Left, const year& Right ) noexcept; // C++ 20

constexpr strong_ordering operator<=>(const year_month& Left, const year_month& Right) noexcept; // C++ 20

template<class Clock, class Duration1, three_way_comparable_with<Duration1> Duration2>
    constexpr auto operator<=>(const time_point<Clock, Duration1>& Left, const time_point<Clock, Duration2>& Right); // C++ 20

template<class Rep1, class Period1, class Rep2, class Period2>
  requires three_way_comparable<typename CT::rep>
    constexpr auto operator<=>(const duration<Rep1, Period1>& Left, const duration<Rep2, Period2>& Right);

2)
constexpr strong_ordering operator<=>(const month_day_last& Left, const month_day_last& Right) noexcept;

3)
```

```
constexpr strong_ordering operator<=>(const year_month_day_last& Left, const
year_month_day_last& Right) noexcept;

4)
strong_ordering operator<=>(const time_zone_link& Left, const
time_zone_link& Right) noexcept;
```

Parameters

Left, *Right*

The day, duration, month, month_day, month_day_last, time_point, time_zone_link, year, year_month, year_month_day, year_month_day_last to compare.

Return value

1)

`0` if `Left == Right`

`< 0` if `Left < Right`

`> 0` if `Left > Right`

2)

Equivalent to: `Left.month() <=> Right.month()`

3)

Equivalent to:

```
C++

if (auto c = Left.year() <=> Right.year(); c != 0) return c;
return Left.month_day_last() <=> Right.month_day_last();
```

4)

Equivalent to:

```
C++

Left.name() <=> Right.name()
```

Example: `operator<=>`

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono; // for day and 'd' literals

int main()
{
    day d1{3};
    day d2{2};

    if ((d1 <=> d2) == 0)
    {
        std::cout << "equal\n";
    }
    else if ((d1 <=> d2) < 0)
    {
        std::cout << "d1 < d2\n";
    }
    else if ((d1 <=> d2) > 0)
    {
        std::cout << "d1 > d2\n";
    }

    std::cout << std::boolalpha << (1d <= 1d) << ' ' << (1d != 2d) << ' ' <<
(2d > 3d);

    return 0;
}
```

Output

```
d1 < d2
true true false
```

operator<<

Output the following types to a stream:

- day
- file_time
- gps_time
- hh_mm_ss
- local_time
- local_info
- month
- month_day

- month_day_last
- month_weekday
- month_weekday_last
- sys_info
- tai_time
- utc_time
- weekday
- weekday_indexed
- weekday_last
- year
- year_month
- year_month_day
- year_month_day_last
- year_month_weekday
- year_month_weekday_last
- zoned_time

C++

```
// 1) day
template <class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os, const day& d); // C++ 20

// 2) hh_mm_ss
template<class CharT, class traits, class Duration>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const hh_mm_ss<Duration>& hms);
// C++ 20

// 3) month
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const month& m); // C++ 20

// 4) month_day
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const month_day& md); // C++ 20

// 5) month_day_last
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const month_day_last& mdl); //
C++ 20

// 6) month_weekday
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
```

```
operator<<(basic_ostream<CharT, Traits>& os, const month_weekday& mwd); //
C++ 20

// 7) month_weekday_last
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const month_weekday_last&
mwdl); // C++ 20

// 8) weekday
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const weekday& wd); // C++ 20

// 9) weekday_indexed
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const weekday_indexed& wdi); //
C++ 20

// 10) weekday_last
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const weekday_last& wdl); //
C++ 20

// 11) year
template <class CharT, class Traits>
std::basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year& y); // C++ 20

// 12) year_month
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year_month& ym); // C++
20

// 13) year_month_day
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year_month_day& ymd); //
C++ 20

// 14) year_month_day_last
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year_month_day_last&
ymdl); // C++ 20

// 15) year_month_weekday
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year_month_weekday&
ymwd); // C++ 20
```

```
// 16) year_month_weekday_last
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const year_month_weekday_last&
ymwdl); // C++ 20

// 17) tai_time
template<class CharT, class Traits, class Duration>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const tai_time<Duration>& t);
// C++ 20

// 18) utc_time
template<class CharT, class Traits, class Duration>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, Traits>& os, const utc_time<Duration>& t);
// C++ 20

// 19) gps_time
template<class CharT, class Traits, class Duration>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const gps_time<Duration>& t);
// C++ 20

// 20) local_time
template<class CharT, class Traits, class Duration>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const local_time<Duration>& t);
// C++ 20

// 21) sys_info
template<class CharT, class Traits>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const sys_info& si);

// 22) local_info
template<class CharT, class Traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<CharT, Traits>& os, const local_info& li);

// 23) zoned_time
template<class CharT, class Traits, class Duration, class TimeZonePtr>
basic_ostream<CharT, Traits>&
operator<<(basic_ostream<CharT, Traits>& os, const zoned_time<Duration,
TimeZonePtr>& zt);
```

Parameters

CharT

The data type of a single character to be read from the stream and stored in the string. The C++ Standard Library provides specializations of this class template, with the type

definitions string for elements of type `char`, wstring, for `wchar_t`, u16string for `char16_t`, and u32string for `char32_t`.

`Traits`

Describes `CharT` attributes for the `basic_string` and `basic_istream` specialization.

`os`

The output stream to emit the `day` value into.

`d`

The `day` to output.

`hms`

The `hh_mm_ss` to output.

`li`

The `local_info` to output.

`m`

The `month` to output.

`md`

The `month_day` to output.

`mdl`

The `month_day_last` to output.

`mwd`

The `month_weekday` to output.

`mwdl`

The `month_weekday_last` to output.

`si`

The `sys_info` to output.

`t`

The `local_time`, `gps_time`, `tai_time`, or `utc_time` to output.

`TimeZonePtr`

A pointer to the time_zone stored in the zoned_time.

wd
The `weekday` to output.

wdi
The `weekday_indexed` to output.

wdl
The `weekday_last` to output.

y
The `year` to output.

ym
The `year_month` to output.

ymd
The `year_month_day` to output.

ymdl
The `year_month_day_last` to output.

ymwd
The `year_month_weekday` to output.

ymwdl
The `year_month_weekday_last` to output.

zt
The `zoned_time` to output.

Return value

The output stream you passed in, `os`

Remarks

1) The `day` value is output as a decimal number, with a leading zero if the result would be a single digit. If `!d.ok()`, then " isn't a valid day" is appended to the output.

2) The `hh_mm_ss` value is output as hours:minutes:seconds:thousandths of seconds. For example, `"00:00:05.721"`

3) The abbreviated month name, using the locale associated with `os`, is output. For example, `Jan`. If `!m.ok()`, then `" is not a valid month"` is appended to the output.

4) The abbreviated month name, using the locale associated with `os`, followed by the date, with a leading zero if the result would be a single digit, is output. For example, `Jan/05`. If `!md.ok()`, then `" is not a valid month"` may be appended to the output of the month, and `"is not a valid day"` may be appended to the output of the day. For example, `204 is not a valid month/204 is not a valid day`.

5) The abbreviated month name, using the locale associated with `os`, followed by `/last`. For example, `Jan/last`.

6) The abbreviated weekday name, using the locale associated with `os`, followed by the nth weekday in the month it represents in brackets. For example, `Mon[1]`.

7) The abbreviated weekday name, using the locale associated with `os`, followed by the last weekday in the month it represents in brackets. For example, `Jan/Mon[last]`.

8) The abbreviated weekday name, using the locale associated with `os`, is output. If `!wd.ok()`, then `" is not a valid weekday"` is appended to the output.

9) The abbreviated weekday name, using the locale associated with `os`, is output, followed by the weekday of the month in brackets. For example, `Mon[3]`. If `!wd.ok()`, then `" is not a valid weekday"` may be appended to the day of the week output, and `"is not a valid index"` may be appended to the weekday index output.

10) The last weekday of a month, using the locale associated with `os`, is output, followed by `[last]`, followed by the date. For example, `Tue[last] 2019-10-29`. If `!wd.ok()`, then `" is not a valid weekday"` may be appended to the day of the week output, and `"is not a valid index"` may be appended to the weekday index output.

11) The year is left-padded with 0 (zero) to four digits if the result would be less than that. If `!y.ok()`, then `" is not a valid year"` is appended to the output.

12) The `year_month` is output in the form yyy-mm-dd. If `ym.ok` returns `false`, then `" is not a valid date"` is appended.

13) The `year_month_day` is output in the form yyyy-mm-dd. If `ymd.ok` returns `false`, then `" is not a valid date"` is appended.

14) The `year_month_day_last` is output in the form yyyy/month/last. For example, `2020/May/last`.

15) The `year_month_weekday` is output in the form yyyy/month/weekday[index]. For example, `1996/Jan/Wed[1]`

16) The `year_month_weekday_last` is output in the form yyyy/month/weekday[last]. For example, `1996/Jan/Wed[last]`

17) The `tai_time` is output in the form yyyy-mm-dd hh:mm:ss.sssssss. For example, `2021-08-13 23:23:08.4358666`

18) The `utc_time` is output in the form yyyy-mm-dd hh:mm:ss.sssssss. For example, `2021-08-13 23:23:08.4358666`

19) The `gps_time` is output in the form yyyy-mm-dd hh:mm:ss.sssssss. For example, `2021-08-13 23:23:08.4358666`

20) The `local_time` is output as the number of seconds since the clock's epoch. It's output as though by `os << std::chrono::sys_time<Duration> (some_local_time.time_since_epoch());`. For example, if `some_local_time` is August 18, 2021 3:13pm, the output is `1597792380`.

21) In Microsoft's implementation, a `sys_info` is output as its `begin`, `end`, `offset`, `save`, and `abbrev` fields. For example: `begin: 2021-03-14 10:00:00, end: 2021-11-07 09:00:00, offset: -25200s, save: 60min, abbrev: PDT`

22) In Microsoft's implementation, a `local_info` is output as yyyy-mm-dd hh:mm::ss.ssssss. For example, `2021-09-17 13:55:59.6590120`

23) The local time in the `zoned_time` (obtained as `zt.get_local_time()`) is output using the format yyyy-mm-dd hh:mm:ss timezone. For example, `2021-09-15 10:45:00 GMT-6`

Example: `operator<<`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    std::cout << utc_clock::now() << '\n';

    year_month ym{ 2021y / April };
```

```
    std::cout << ym;
    return 0;
}
```

Output

```
2021-08-16 20:47:05.6299822
2021/Apr
```

`operator modulo`

Operator for modulo operations on duration.

C++

```
1)
template <class Rep1, class Period1, class Rep2>
constexpr duration<Rep1, Period1, Rep2>::type
   operator%(
      const duration<Rep1, Period1>& Dur,
      const Rep2& Div);

2)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<duration<Rep1, _Period1>, duration<Rep2,
Period2>>::type
   operator%(
     const duration<Rep1, Period1>& Left,
     const duration<Rep2, Period2>& Right);
```

Parameters

`Dur`

A `duration` object.

`Div`

An integral value.

`Left`

The dividend. The modulo is the remainder after dividing the dividend by the divisor.

`Right`

The right `duration` object, the divisor.

Return value

1) Returns a `duration` object whose interval length is *`Dur`* modulo *`Div`*.

2) Returns a value that represents *`Left`* modulo *`Right`*.

`operator/` for `duration`

Division operator for duration objects.

C++

```
1)
template <class Rep1, class Period1, class Rep2>
constexpr duration<typename common_type<Rep1, Rep2>::type, Period1>
   operator/(
     const duration<Rep1, Period1>& Dur,
     const Rep2& Div);

2)
template <class Rep1, class Period1, class Rep2, class Period2>
constexpr typename common_type<Rep1, Rep2>::type
   operator/(
     const duration<Rep1, Period1>& Left,
     const duration<Rep2, Period2>& Right);
```

Parameters

`Dur`
A `duration` object.

`Div`
An integral value.

`Left`\w The left `duration` object.

`Right`
The right `duration` object.

Return value

1) A duration object whose interval length is the length of *`Dur`* divided by the value *`Div`*.

2) The ratio of the interval lengths of *`Left`* and *`Right`*.

Unless `is_convertible<Rep2, common_type<Rep1, Rep2>>` *holds true*, and `Rep2` isn't an instantiation of `duration`, the first operator doesn't participate in overload resolution. For more information, see <type_traits>.

`operator/` for calendar dates

Provides syntax to create calendar dates in the following forms:

month/day/year
day/month/year
year/month/day

You can replace *day* with:

`last`
`weekday[n]` for the nth day of the month
`weekday[last]` for the last weekday of the month.

Partial dates can be formed as follows:

C++

```
year_month ym = 2015y/April;
month_day md1 = April/4;
month_day md2 = 4d/April;
```

Integers can be used as long as the interpretation isn't ambiguous.

C++

```
/////////  returns year_month

// 1
constexpr year_month
operator/(const year& y, const month& m) noexcept; // C++ 20

// 2
constexpr year_month
operator/(const year& y, int m) noexcept; // C++ 20

/////////  returns month_day

// 3
constexpr month_day
operator/(const month& m, const day& d) noexcept; // C++ 20

// 4
```

```
constexpr month_day
operator/(const month& m, int d) noexcept; // C++ 20

// 5
constexpr month_day
operator/(int m, const day& d) noexcept; // C++ 20

// 6
constexpr month_day
operator/(const day& d, const month& m) noexcept; // C++ 20

// 7
constexpr month_day
operator/(const day& d, int m) noexcept; // C++ 20

/////////  returns month_day_last

// 8
constexpr month_day_last
operator/(const month& m, last_spec) noexcept; // C++ 20

// 9
constexpr month_day_last
operator/(int m, last_spec) noexcept; // C++ 20

// 10
constexpr month_day_last
operator/(last_spec, const month& m) noexcept; // C++ 20

// 11
constexpr month_day_last
operator/(last_spec, int m) noexcept; // C++ 20

/////////  returns month_weekday

// 12
constexpr month_weekday
operator/(const month& m, const weekday_indexed& wdi) noexcept; // C++ 20

// 13
constexpr month_weekday
operator/(int m, const weekday_indexed& wdi) noexcept; // C++ 20

// 14
constexpr month_weekday
operator/(const weekday_indexed& wdi, const month& m) noexcept; // C++ 20

// 15
constexpr month_weekday
operator/(const weekday_indexed& wdi, int m) noexcept; // C++ 20

/////////  returns month_weekday_last

// 16
constexpr month_weekday_last
```

```
operator/(const month& m, const weekday_last& wdl) noexcept; // C++ 20

// 17
constexpr month_weekday_last
operator/(int m, const weekday_last& wdl) noexcept; // C++ 20

// 18
constexpr month_weekday_last
operator/(const weekday_last& wdl, const month& m) noexcept; // C++ 20

// 19
constexpr month_weekday_last
operator/(const weekday_last& wdl, int m) noexcept; // C++ 20

/////////  returns year_month_day

// 20
constexpr year_month_day
operator/(const year_month& ym, const day& d) noexcept; // C++ 20

// 21
constexpr year_month_day
operator/(const year_month& ym, int d) noexcept; // C++ 20

// 22
constexpr year_month_day
operator/(const year& y, const month_day& md) noexcept; // C++ 20

// 23
constexpr year_month_day
operator/(int y, const month_day& md) noexcept; // C++ 20

// 24
constexpr year_month_day
operator/(const month_day& md, const year& y) noexcept; // C++ 20

// 25
constexpr year_month_day
operator/(const month_day& md, int y) noexcept; // C++ 20

/////////  returns year_month_day_last

// 26
constexpr year_month_day_last
operator/(const year_month& ym, last_spec) noexcept; // C++ 20

// 27
constexpr year_month_day_last
operator/(const year& y, const month_day_last& mdl) noexcept; // C++ 20

// 28
constexpr year_month_day_last
operator/(int y, const month_day_last& mdl) noexcept; // C++ 20

// 29
```

```
constexpr year_month_day_last
operator/(const month_day_last& mdl, const year& y) noexcept; // C++ 20

// 30
constexpr year_month_day_last
operator/(const month_day_last& mdl, int y) noexcept; // C++ 20

/////////  returns year_month_weekday

// 31
constexpr year_month_weekday
operator/(const year_month& ym, const weekday_indexed& wdi) noexcept; // C++
20

// 32
constexpr year_month_weekday
operator/(const year& y, const month_weekday& mwd) noexcept; // C++ 20

// 33
constexpr year_month_weekday
operator/(int y, const month_weekday& mwd) noexcept; // C++ 20

// 34
constexpr year_month_weekday
operator/(const month_weekday& mwd, const year& y) noexcept; // C++ 20

// 35
constexpr year_month_weekday
operator/(const month_weekday& mwd, int y) noexcept; // C++ 20

/////////  returns year_month_weekday_last

// 36
constexpr year_month_weekday_last
operator/(const year_month& ym, const weekday_last& wdl) noexcept; // C++ 20

// 37
constexpr year_month_weekday_last
operator/(const year& y, const month_weekday_last& mwdl) noexcept; // C++ 20

// 38
constexpr year_month_weekday_last
operator/(int y, const month_weekday_last& mwdl) noexcept; // C++ 20

// 39
constexpr year_month_weekday_last
operator/(const month_weekday_last& mwdl, const year& y) noexcept; // C++ 20

// 40
constexpr year_month_weekday_last
operator/(const month_weekday_last& mwdl, int y) noexcept; // C++ 20
```

Parameters

`d`

The day. Provided either as an integer in the range [1,31], or as a day.

`lastspec`

An empty tag type that indicates the last item in s sequence. For example, `2021y/May/last` is the last day of May 2021.

`m`

The month. Provided either as an integer in the range [1,12], or as a month.

`md`

The month and day.

`mdl`

The last day of the specified month.

`mwd`

The n-th weekday of the specified month.

`mwdl`

The last weekday of the specified month.

`wdi`

A weekday index (weekday_indexed). For example, `weekday_indexed(Monday, 1)` is the first Monday of a month.

`wdl`

The last weekday of a month. For example, `Monday[last]` is the last Monday of a month.

`y`

The year. Provided either as an integer, or as a year.

`ym`

The year and month.

Return value

1) `year_month(y, m)`
2) `year_month(y, month(m))`
3) `month_day(m, d)`
4) `month_day(m, day(d))`
5) `month_day(month(m), d)`
6) `month_day(m, d)`

7) `month_day(month(m), d)`
8) `month_day_last(m)`
9) `month_day_last(month(m))`
10) `month_day_last(m)`
11) `month_day_last(month(m))`
12) `month_weekday(m, wdi)`
13) `month_weekday(month(m), wdi)`
14) `month_weekday(m, wdi)`
15) `month_weekday(month(m), wdi)`
16) `month_weekday_last(m, wdl)`
17) `month_weekday_last(month(m), wdl)`
18) `month_weekday_last(m, wdl)`
19) `month_weekday_last(month(m), wdl)`
20) `year_month_day(ym.year(), ym.month(), d)`
21) `year_month_day(ym.year(), ym.month(), day(d))`
22) `year_month_day(y, md.month(), md.day())`
23) `year_month_day(year(y), md.month(), md.day())`
24) `year_month_day(y, md.month(), md.day())`
25) `year_month_day(year(y), md.month(), md.day())`
26) `year_month_day_last(ym.year(), month_day_last(ym.month()))`
27) `year_month_day_last(y, mdl)`
28) `year_month_day_last(year(y), mdl)`
29) `year_month_day_last(y, mdl)`
30) `year_month_day_last(year(y), mdl)`
31) `year_month_weekday(ym.year(), ym.month(), wdi)`
32) `year_month_weekday(y, mwd.month(), mwd.weekday_indexed())`
33) `year_month_weekday(year(y), mwd.month(), mwd.weekday_indexed())`
34) `year_month_weekday(y, mwd.month(), mwd.weekday_indexed())`
35) `year_month_weekday(year(y), mwd.month(), mwd.weekday_indexed())`
36) `year_month_weekday_last(ym.year(), ym.month(), wdl)`
37) `year_month_weekday_last(y, mwdl.month(), mwdl.weekday_last())`
38) `year_month_weekday_last(year(y), mwdl.month(), mwdl.weekday_last())`
39) `year_month_weekday_last(y, mwdl.month(), mwdl.weekday_last())`
40) `year_month_weekday_last(year(y), mwdl.month(), mwdl.weekday_last())`

Example: `operator/` for calendar dates

C++

```cpp
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    month m{ July }; // Jul
    month_day md{ April / 4 }; // Apr/04
    month_day md2{ 4d / April }; // Apr/04
    month_day_last mdl{ January / last }; // Jan/last
    month_weekday mw{ 11 / Monday[1] }; // Nov/Mon[1]
    month_weekday_last mwl{ January / Monday[last] }; // Jan/Mon[last]
    weekday wd{ Monday }; // Mon
    weekday_indexed wdi{ Monday, 1 }; // Mon[1]
    year_month ym{ 2021y / April }; // 2021/Apr
    year_month_day ymd{ January / 1d / 2021y }; // 2021-01-01
    year_month_day ymd2{ 2021y / 5 / 7 }; // 2021-05-07
    year_month_day_last ymdl{ April / last / 1975 }; // 1975/Apr/last
    year_month_weekday ymw{ 1997y / January / Wednesday[1] }; //
1997/Jan/Wed[1]
    year_month_weekday_last ymwl{ 1997y / January / Wednesday[last] }; //
1997/Jan/Wed[last]
    int yearValue{ 2021 / 4 / 4 }; // 126

    std::cout << m << '\n' << md << '\n' << md2 << '\n' << mdl << '\n' << mw
        << '\n' << mwl << '\n' << wd << '\n' << wdi << '\n'
        << ym << '\n' << ymd << '\n' << ymd2 << '\n' << ymdl
        << '\n' << ymw << '\n' << ymwl << '\n' << yearValue;

    return 0;
}
```

Output

```
Jul
Apr/04
Apr/04
Jan/last
Nov/Mon[1]
Jan/Mon[last]
Mon
Mon[1]
2021/Apr
2021-01-01
2021-05-07
1975/Apr/last
1997/Jan/Wed[1]
1997/Jan/Wed[last]
126
```

`chrono` literals

Article • 08/03/2021

(C++14) The `<chrono>` header defines 12 user-defined literals that represent hours, minutes, seconds, milliseconds, microseconds, and nanoseconds. Each user-defined literal has an integral and a floating-point overload. The literals are defined in the `literals::chrono_literals` inline namespace, which is brought into scope automatically when `std::chrono` is in scope.

Syntax

C++

```
inline namespace literals {
  inline namespace chrono_literals {
    // return integral days
    constexpr std::chrono::day operator"" d(unsigned long long d) noexcept;

    // return integral hours
    constexpr chrono::hours operator"" h(unsigned long long Val);

    // return floating-point hours
    constexpr chrono::duration<double, ratio<3600>> operator"" h(long double
Val);

    // return integral minutes
    constexpr chrono::minutes(operator"" min)(unsigned long long Val);

    // return floating-point minutes
    constexpr chrono::duration<double, ratio<60>>(operator"" min)(long
double Val);

    // return integral seconds
    constexpr chrono::seconds operator"" s(unsigned long long Val);

    // return floating-point seconds
    constexpr chrono::duration<double> operator"" s(long double Val);

    // return integral milliseconds
    constexpr chrono::milliseconds operator"" ms(unsigned long long Val);

    // return floating-point milliseconds
    constexpr chrono::duration<double, milli> operator"" ms(long double
Val);

    // return integral microseconds
    constexpr chrono::microseconds operator"" us(unsigned long long Val);
```

```
    // return floating-point microseconds
    inline constexpr chrono::duration<double, micro> operator"" us(long
double Val);

    // return integral nanoseconds
    inline constexpr chrono::nanoseconds operator"" ns(unsigned long long
Val);

    // return floating-point nanoseconds
    constexpr chrono::duration<double, nano> operator"" ns(long double Val);

    // return integral year
    constexpr chrono::year operator""y(unsigned long long y) noexcept; //
C++ 20
  } // inline namespace chrono_literals
} // inline namespace literals
```

Return value

The literals that take a **`long long`** argument return a value or the corresponding type.

The literals that take a floating point argument return a duration.

Example

The following examples show how to use the `chrono` literals.

C++

```
constexpr auto day = 14d; // If the value > 255, then the stored value is
unspecified.
constexpr auto twoDays = 48h;
constexpr auto week = 24h* 7;
constexpr auto my_duration_unit = 108ms;
constexpr auto theYear = 2021y;
```

`ambiguous_local_time` class

Article • 06/15/2022

This exception is thrown when attempting to convert a `local_time` to a `sys_time` and the result can be one of two times and neither `choose::earliest` or `choose::latest` was specified to settle the ambiguity.

Syntax

C++

```
class ambiguous_local_time : public runtime_error; // C++ 20
```

Remarks

During the transition from daylight saving time to standard time in the fall, clocks essentially gain an extra hour. This can be confusing because doesn't the transition to standard time mean losing an hour? By falling back an hour, the hour before the transition will be repeated after the clock adjusts for standard time. Consider the change to standard time in New York, which happens on the first Sunday in November at 2:00am. First, 1:00am goes by. At 2am, the clock transitions to standard time, so now it's 1:00am again. That means the time between 1am and 2am will be "repeated", effectively adding an hour.

If a `local_time` specifies a time during this "extra" hour, it isn't clear how to convert it. Should the converted time be treated as the "first" time that hour happens, or the "second"? If the enum choose isn't specified to indicate which it should be, you'll get an ambiguous_local_time exception.

This problem doesn't exist when converting from standard time to daylight saving time. In that case, a different problem can arise. See nonexistent_local_time for details.

The following example shows an ambiguous conversion.

Example: `ambiguous_local_time`

C++

```
#include <chrono>
#include <iostream>
```

```
using namespace std::chrono;

int main()
{
    try
    {
        // The following will throw an exception because converting 1:30am
local time to system time could be interpreted as either
        // 1:30 AM EDT or 1:30 AM EST. Which to choose isn't specified for
the conversion, so an ambiguous_local_time
        // exception is thrown.
        auto zt = zoned_time{"America/New_York",
local_days{Sunday[1]/November/2016} + 1h + 30min};
    } catch (const ambiguous_local_time& e)
    {
        std::cout << e.what() << '\n';
    }
    return 0;
}
```

Output

```
2016-11-06 01:30:00 is ambiguous. It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC
```

Members

Name	Description
Constructor	Construct an `ambiguous_local_time`.
what	Gets a string that describes the nature of the ambiguity.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Constructs a `ambiguous_local_time`.

```
C++
```

```
template<class Duration>
ambiguous_local_time(const local_time<Duration>& tp, const local_info& i);
```

Parameters

tp

The `local_time` that couldn't be converted.

i

Information about the attempted conversion. See local_info for details.

Remarks

You typically won't create this exception. It's thrown by functions that convert a `local_time` to a `sys_time`.

`what`

Gets a string describing the details of the ambiguity.

```
C++
```

```
[nodiscard] virtual const char* what() const noexcept;
```

Return value

A string describing the ambiguity. For example:

```
Output
```

```
2016-11-06 01:30:00 is ambiguous. It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC
```

See also

<chrono>
to_sys
nonexistent_local_time
Header Files Reference

`choose` enum

Article • 06/15/2022

Used with time_zone and zoned_time to indicate how to handle an ambiguous or nonexistent `local_time` to `sys_time` conversion.

Syntax

```
C++
```

```
enum class choose { // C++ 20
    earliest,
    latest
};
```

Members

Element	Description
`earliest`	If a time conversion is ambiguous, uses the earlier time point.
`latest`	If a time conversion is ambiguous, uses the later time point.

Remarks

If the result of the conversion is a `nonexistent_local_time`, the time in `earliest` and `latest` are the same.

Ambiguous conversion error

During the transition from daylight saving time to standard time in the fall, clocks essentially gain an extra hour. This can be confusing because doesn't the transition to standard time mean losing an hour? By falling back an hour, the hour before the transition will be repeated after the clock adjusts for standard time. Consider the change to standard time in New York, which happens on the first Sunday in November at 2:00am. First, 1:00am goes by. At 2am, the clock transitions to standard time, so now it's 1:00am again. That means the time between 1am and 2am will be "repeated", effectively adding an hour.

If a `local_time` specifies a time during this "extra" hour, it isn't clear how to convert it. Should the converted time be treated as the "first" time that hour happens, or the "second"? If the enum choose isn't specified to indicate which it should be, you'll get an `ambiguous_local_time` exception.

Nonexistent conversion error

When converting from standard time to daylight saving time, a different problem can arise. During the transition from standard time to daylight saving time in the spring, clocks move ahead an hour. The skipped hour disappears, so even though it appears we've added an hour to the time, by "springing forward" an hour, the hour following the transition is effectively removed.

Consider the change to daylight saving time in New York, which happens on the second Sunday in March at 2am. At 2am, the clock transitions to daylight savings time and now reads 3:00am. If the `local_time` being converted is 2:30am, for example, that time is during the period that was "removed" and so is "nonexistent", resulting in a nonexistent_local_time exception.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler option:/std:c++latest is required.

See also

to_sys
zoned_time constructor

`clock_time_conversion` struct

Article • 06/15/2022

A trait that specifies how to convert a time_point from one type of clock to another, while maintaining the equivalent point in time.

Syntax

C++

```
// C++20
1) template<class Dest, class Source> struct clock_time_conversion {};
2) template<class Clock> struct clock_time_conversion<Clock, Clock>;
3) template<> struct clock_time_conversion<system_clock, system_clock>;
4) template<> struct clock_time_conversion<utc_clock, utc_clock>;
5) template<> struct clock_time_conversion<system_clock, utc_clock>;
6) template<> struct clock_time_conversion<utc_clock, system_clock>;
7) template<class Clock> struct clock_time_conversion<Clock, system_clock>;
8) template<class Clock> struct clock_time_conversion<system_clock, Clock>;
9) template<class Clock> struct clock_time_conversion<Clock, utc_clock>;
10) template<class Clock> struct clock_time_conversion<utc_clock, Clock>;
```

Template parameters

`Clock`

A clock type to convert from/to.

`Dest`

The clock type to convert to.

`Source`

The clock type to convert from.

The traits provide the following conversions:

1) An empty struct that is defined only so that it can be specialized.
2-4) Identity conversions. Returns the same clock you pass in.
5-6) Converting between `sys_time` and `utc_time` calls `utc_clock::to_sys` or `utc_clock::from_sys` depending on the direction of the conversion.
7-8) Converting between `sys_time` and the specified clock, when the specified clock supports `to_sys` and `from_sys`, results in a call to `Clock::to_sys` or `Clock::from_sys`, depending on the direction of the conversion.

9-10) Converting between `utc_time` and the specified clock, when the specified clock supports `from_utc` and `to_sys`, results in a call to `Clock::to_utc` or `Clock::from_utc`, depending on the direction of the conversion.

Members

Name	Description
operator ()	Converts a `time_point` from one clock to another.

Remarks

You usually won't use this trait directly in your code. It's used by the clock_cast conversion function.

Requirements

Header: <`chrono`>

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`operator()`

Converts a `time_point` from one clock type to another while maintaining the equivalent point in time.

Syntax

```
C++

1)
template <class Duration>
time_point<Clock, Duration> operator()(const time_point<Clock, Duration>& t)
const;

2)
template <class Duration>
sys_time<Duration> operator()(const sys_time<Duration> & t) const;

3)
```

```
template <class Duration>
utc_time<Duration> operator()(const utc_time<Duration>& t) const;

4)
template <class Duration>
sys_time<Duration> operator()(const utc_time<Duration>& t) const;

5)
template <class Duration>
utc_time<Duration> operator()(const sys_time<Duration>& t) const;
```

Parameters

`t` The `time_point` to convert.

Return value

1-3) Identity conversions. No conversion. Returns `t` without any changes.

4) Returns `utc_clock::to_sys(t)`.

5) Returns `utc_clock::from_sys(t)`.

Deduction guides

The following deduction guides are provided for `template <class Duration> operator()`:

C++

```
1)
template <class Duration> auto operator()(const sys_time<Duration>& t) const
    -> decltype(Clock::from_sys(t));

2)
template <class Duration> auto operator()(const time_point<SourceClock,
Duration>& t) const
    -> decltype(Clock::to_sys(t));

3)
template <class Duration> auto operator()(const utc_time<Duration>& t) const
    -> decltype(Clock::from_utc(t));

4)
template <class Duration> auto operator()(const time_point<Clock, Duration>&
t) const
    -> decltype(Clock::to_utc(t));
```

1) Participates in overload resolution only when `Clock` supports `from_sys()` and returns `time_point<Clock, Duration>`.
2) Participates in overload resolution only when `Clock` supports `to_sys()` and returns `sys_time<Duration>`.
3) Participates in overload resolution only when `Clock` supports `from_utc()` and returns `time_point<Clock, Duration>`.
4) Participates in overload resolution only when `Clock` supports `to_utc()` and returns `utc_time<Duration>`.

Example: `clock_time_conversion`

C++

```
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    auto sd = sys_days{ 2021y / July / 26 };
    auto time = clock_time_conversion<utc_clock, system_clock>{}(sd);
    std::cout << time << "\n";
    return 0;
}
```

Output

```
2021-07-26 00:00:00
```

See also

<chrono>
clock_cast
Header Files Reference

`common_type` Structure

Article • 06/15/2022

Describes specializations of class template common_type for instantiations of duration and time_point.

Syntax

```
C++
```

```
template <class Rep1, class Period1,
    class Rep2, class Period2>
struct common_type
<chrono::duration<Rep1, Period1>,
chrono::duration<Rep2, Period2>>;

template <class Clock,
    class Duration1, class Duration2>
struct common_type
<chrono::time_point<Clock, Duration1>,
chrono::time_point<Clock, Duration2>>;
```

Requirements

Header: `<chrono>`

Namespace: `std`

See also

<chrono>
Header Files Reference

`day` class

Article • 06/15/2022

Represents a day of the month. For example, the 25th day of the month.

Syntax

```
C++
```

```
class day; // C++ 20
```

Remarks

A `day` normally holds values in the range [1, 31]. It may hold non-negative values outside this range, but the behavior is unspecified if it isn't within the range [0, 255].

Members

Name	Description
Constructors	Construct a `day`.
ok	Verify that the day value is in the valid range [1,31].
operator++	Increment this `day`.
operator+=	Add the specified number of days to this `day`.
operator--	Decrement the `day`.
operator-=	Subtract the specified number of days from this `day`
operator unsigned	Get the day value as an unsigned integer.

Non-members

Name	Description
from_stream	Parse a `day` from the given stream using the specified format.
operator+	Add specified number of days to this `day`, returning a new `day` object.

Name	Description
operator-	Subtract the specified number of days from this `day`, returning a new `day` object.
operator==	Determine whether two `day` instances are equal.
operator<=>	Compare this `day` against another `day`. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `day` to the given stream.
operator""d	Create a `day` literal for a day in the month.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Constructs a `day`.

```
C++

1) day() = default;
2) explicit constexpr day(unsigned d) noexcept;
```

Parameters

d

Construct a `day` with value *d*.

Remarks

1) The default constructor doesn't initialize the day value.
2) Construct a `day` with the day value initialized to `d`.

Example: Create a `day`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    day d{1}; // day 1
    day d2 = 2d; // day 2

    std::cout << d << ", " << d2;

    return 0;
}
```

Output

```
01, 02
```

ok

Check if the value stored in this `day` is in the valid range.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the day value is in the range [1,31]. Otherwise, `false`.

operator++

Add 1 to the day value.

C++

```
1) constexpr std::chrono::day& operator++() noexcept;
2) constexpr std::chrono::day operator++(int) noexcept;
```

Return value

1) A reference to `*this` `day` *after* it has been incremented (a postfix increment).
2) A copy of the `day`, *before* it has been incremented (a prefix increment).

Example: `operator++`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    day d{1};

    std::cout << d << " " << ++d << "\n"; // constexpr day& operator++()
noexcept
    std::cout << d << " " << d++ << "\n"; // constexpr day operator++(int)
noexcept
    std::cout << d;

    return 0;
}
```

Output

```
01 02
02 02
03
```

Remarks

If the incremented result is out of the range [0, 255], then the stored value is unspecified.

`operator--`

Subtract 1 from the day value.

C++

```
1) constexpr std::chrono::day& operator--() noexcept;
2) constexpr std::chrono::day operator--(int) noexcept;
```

Return value

1) A reference to `*this` day *after* it has been decremented (a postfix decrement).
2) A copy of the `*day`, *before* it has been decremented (a prefix decrement).

Example: `operator--`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

int main()
{
    day d{5};
    std::cout << d << " " << --d << "\n"; // constexpr day& operator--()
noexcept
    std::cout << d << " " << d-- << "\n"; // constexpr day operator--(int)
noexcept
    std::cout << d << "\n";

    return 0;
}
```

Output

```
05 04
04 04
03
```

Remarks

If the decremented result is out of the range [0, 255], then the stored value is unspecified.

`operator+=`

Add days to this `day`.

```
C++
```

```
constexpr day& operator+=(const days& d) noexcept;
```

Parameters

`d`

The number of days to add.

Return value

`*this`

If the result is outside the range [0, 255], then the stored value is unspecified.

`operator-=`

Subtract days from this `day`.

```
C++
```

```
constexpr day& operator-=(const days& d) noexcept;
```

Parameters

`d`

The number of days to subtract.

Return value

`*this`. If the result is outside the range [0, 255], then the stored value is unspecified.

`operator unsigned`

Get the `day` value.

```
C++
```

```
explicit constexpr operator unsigned() const noexcept;
```

Return value

The value of the `day`

Example: `operator unsigned()`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

int main()
{
    chrono::day d{14d};
    unsigned dayValue = static_cast<unsigned>(d);
    cout << dayValue << "\n";

    return 0;
}
```

Output

```
14
```

See also

<chrono>
month_day class
month_day_last class
year_month_day
year_month_day_last
Header Files Reference

`duration` class

Article • 06/15/2022

Measures a time span such as one minute, two hours, ten milliseconds, and so on.

A `duration` holds a *time interval*, which is the number of ticks over a time unit. For example, five minutes is five ticks, with each tick a minute apart. 42 seconds is 42 ticks, with each tick a second apart.

Syntax

C++

```
template <class Rep, class Period> class duration;
template <class Rep, class Period = ratio<1>> class duration;
template <class Rep, class Period1, class Period2> class duration
<duration<Rep, Period1>, Period2>;
```

Remarks

The template argument `Rep` describes the type that is used to hold the number of clock ticks in the interval. The template argument `Period` is an instantiation of ratio that describes the size of the interval that each tick represents.

Members

Constructors

Name	Description
duration	Constructs a `duration` object.

Functions

Name	Description
count	Returns the number of clock ticks in the time interval.
max	Static. Returns the maximum allowable value of template parameter `Rep`.

Name	Description
min	Static. Returns the lowest allowable value of template parameter `Rep`.
zero	Static. In effect, returns `Rep(0)`.

Operators

Name	Description
duration::operator-	Returns a copy of the `duration` object with a negated tick count.
duration::operator--	Decrements the stored tick count.
duration::operator-=	Subtracts the tick count of a specified `duration` from the stored tick count.
duration::operator+	Returns `*this`.
duration::operator++	Increments the stored tick count.
duration::operator+=	Adds the tick count of a specified `duration` to the stored tick count.
duration::operator=	Assigns one duration to another.
duration::operator*=	Multiplies the stored tick count by a specified value.
duration::operator/=	Divides the stored tick count by the tick count of a specified `duration` object.
duration::operator%=	Reduces the stored tick count modulo a specified value.

Non-members

Function templates

Name	Description
abs	Returns the absolute value of the `duration`.
ceil	Returns the smallest representable `duration` that's greater than or equal to the specified `duration`.
duration_cast	Casts a `duration` object to a specified target `duration` type.
floor	Returns the greatest representable `duration` that's less than or equal to the specified `duration`.

Name	Description
from_stream	Parse a `duration` from the given stream using the specified format.
round	Rounds the specified `duration` to the nearest representable `duration` in the target type.

Operators

Name	Description
operator+	After converting the durations being added to their common type, returns a `duration` with a tick count equal to the sum of the converted tick counts.
operator-	After converting the durations being subtracted to their common type, returns a `duration` with a tick count equal to the number of ticks in the RHS `duration` subtracted from the number of ticks in the LHS `duration`.
operator*	After converting the durations being multiplied to their common type, returns a `duration` with a tick count equal to the multiplication of the converted tick counts.
operator/	After converting the durations being divided to their common type, returns a `duration` with a tick count equal to the division of the converted tick counts.
operator%	After converting the `duration` and the divisor to their common type, returns a `duration` with a tick count equal to the remainder of the division.
operator==	After converting the `duration` types being compared to their common type, determines if the number of ticks are equal.
operator!=	Determine if `duration` isn't equal to another.
operator<	Determine if one `duration` is less than another.
operator<=	Determine if one `duration` is less than or equal to another.
operator>	Determine if one `duration` is greater than another.
operator>=	Determine if one `duration` is greater than or equal to another.
operator<=>	Compare one `duration` against another `duration`. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `duration` to the given stream.

Convenience type aliases

Name	Description
`duration::period`	A synonym for the template parameter `Period`.
`duration::rep`	A synonym for the template parameter `Rep`.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

`duration::count`

Retrieves the number of clock ticks in the time interval.

C++

```
constexpr Rep count() const;
```

Return value

The number of clock ticks in the time interval.

`duration::duration` constructor

Constructs a `duration` object.

C++

```
1) constexpr duration() = default;
2) constexpr duration(const duration& d) = default;

3) template <class Rep2>
constexpr explicit duration(const Rep2& R);

4) template <class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& Dur);
```

Parameters

Dur

The number of ticks of period specified by *Period2*.

Period2

A `std::ratio` template specialization to represent the tick period in units of seconds.

R

The number of ticks of default period.

Rep2

An arithmetic type to represent the number of ticks.

Remarks

1) The default constructor constructs an object that is uninitialized. Value initialization by using empty braces initializes an object that represents a time interval of zero clock ticks.

2) The copy constructor makes a bitwise copy of `d`.

3) Constructs an object that represents a time interval of *R* clock ticks using a default period of `std::ratio<1>`. To avoid round-off of tick counts, it's an error to construct a duration object from a representation type *Rep2* that can be treated as a floating-point type when `duration::rep` cannot be treated as a floating-point type.

4) Constructs an object that represents a time interval whose length is the time interval that is specified by *Dur*. To avoid truncation of tick counts, it's an error to construct a duration object from another duration object whose type is *incommensurable* with the target type.

A duration type `D1` is *incommensurable* with another duration type `D2` if `D2` cannot be treated as a floating-point type and ratio_divide<D1::period, D2::period>::type::den isn't 1.

Unless *Rep2* is implicitly convertible to `rep` and either `treat_as_floating_point<rep>` *holds true* or `treat_as_floating_point<Rep2>` *holds false*, the second constructor doesn't participate in overload resolution. For more information, see <type_traits>.

Unless no overflow is induced in the conversion and `treat_as_floating_point<rep>` *holds true*, or both `ratio_divide<Period2, period>::den` equals 1 and

`treat_as_floating_point<Rep2>` *holds false*, the third constructor doesn't participate in overload resolution. For more information, see <type_traits>.

Example: Create a `duration`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    // create a duration that tracks ticks as 1/10ths of a second
    duration<int, std::ratio<1, 10>> tenths{ 5 };
    std::cout << tenths << '\n';

    hours h{12}; // hours is a convenience duration typedef
    auto h2 = 3h; // 'h' is a convenience operator. h2 is a duration<int,
std::ratio<3600,1>>

    std::cout << h << ":" << h2 << '\n';

    return 0;
}
```

Output

```
5ds
3h:3h
```

`duration::max`

Static method that returns the upper bound for values of template parameter type `Rep`.

C++

```
static constexpr duration max();
```

Return value

In effect, returns `duration(duration_values<rep>::max())`.

`duration::min`

Static method that returns the lower bound for values of template parameter type `Rep`.

```
C++
```

```
static constexpr duration min();
```

Return value

In effect, returns `duration(duration_values<rep>::min())`.

`duration::operator-`

Returns a copy of the `duration` object with a negated tick count.

```
C++
```

```
constexpr duration operator-() const;
```

`duration::operator--`

Decrements the stored tick count.

```
C++
```

```
1) duration& operator--();
2) duration operator--(int);
```

Return value

1) Returns `*this`. 2) Returns a copy of `*this` before the decrement.

`duration::operator%=`

Reduces the stored tick count modulo the specified value.

```
C++
```

```
1) duration& operator%=(const rep& Div);
2) duration& operator%=(const duration& Div);
```

Parameters

Div

1) *Div* a tick count. 2) *Div* a `duration` that contains a tick count.

Return value

The `duration` object after the modulo operation is done.

`duration::operator*=`

Multiplies the stored tick count by a specified value.

C++

```
duration& operator*=(const rep& Mult);
```

Parameters

Mult

A value of the type that is specified by `duration::rep`.

Return value

The `duration` object after the multiplication is done.

`duration::operator/=`

Divides the stored tick count by a specified value.

C++

```
duration& operator/=(const rep& Div);
```

Parameters

Div

A value of the type that is specified by `duration::rep`.

Return value

The `duration` object after the division is done.

`duration::operator+`

Returns `*this`.

```
C++
```

```
constexpr duration operator+() const;
```

Return value

`*this`

`duration::operator++`

Increments the stored tick count.

```
C++
```

```
1) duration& operator++();
2) duration operator++(int);
```

Return value

1) Returns `*this`. 2) Returns a copy of `*this` before the increment.

`duration::operator+=`

Adds the tick count of a specified `duration` object to the stored tick count.

```
C++
```

```
duration& operator+=(const duration& Dur);
```

Parameters

Dur
A `duration` object.

Return value

The `duration` object after the addition is done.

`duration::operator-=`

Subtracts the tick count of a specified `duration` object from the stored tick count.

```
C++
```

```
duration& operator-=(const duration& Dur);
```

Parameters

Dur
A `duration` object.

Return value

The `duration` object after the subtraction is done.

`duration::zero`

Returns `duration(duration_values<rep>::zero())`.

```
C++
```

```
static constexpr duration zero();
```

`duration::operator=`

Assigns one duration to another.

```
C++
```

```
duration& operator=(const duration &other) = default;
```

Parameters

other

The `duration` object to copy.

Return value

The LHS `duration` object.

See also

<chrono>
Convenience duration typedefs such as minutes, seconds, and more
Convenience literals for hours, minutes, and more
duration_values structure
Header Files Reference

`duration_values` structure

Article • 06/15/2022

Provides specific values for the duration template parameter `Rep`.

Syntax

```
C++
```

```
template <class Rep>
struct duration_values;
```

Members

Public Methods

Name	Description
max	Static. Specifies the upper limit for a value of type `Rep`.
min	Static. Specifies the lower limit for a value of type `Rep`.
zero	Static. Returns `Rep(0)`.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

`duration_values::max`

Static method that returns the upper bound for values of type `Rep`.

```
C++
```

```
static constexpr Rep max();
```

Return value

In effect, returns `numeric_limits<Rep>::max()`.

Remarks

When `Rep` is a user-defined type, the return value must be greater than duration_values::zero.

`duration_values::min`

Static method that returns the lower bound for values of type `Rep`.

```
C++
```

```
static constexpr Rep min();
```

Return value

In effect, returns `numeric_limits<Rep>::lowest()`.

Remarks

When `Rep` is a user-defined type, the return value must be less than or equal to duration_values::zero.

`duration_values::zero`

Returns `Rep(0)`.

```
C++
```

```
static constexpr Rep zero();
```

Remarks

When `Rep` is a user-defined type, the return value must represent the additive infinity.

See also

<chrono>

Header Files Reference

`file_clock` class

Article • 03/08/2024

This clock can represent the range and resolution of file time values used in the file system.

Syntax

```
C++
```

```
using file_clock = std::filesystem::_File_time_clock; // C++20
```

Remarks

In the Microsoft implementation, the epoch, or the time from which the `file_clock` starts measuring time, is 1/1/1601 00:00:00.

The ISO C++ Standard provides a choice between providing `to_sys()` and `from_sys()`, or `to_utc()` and `from_utc()`. The Microsoft implementation chose to_utc and from_utc.

Members

Expand table

Name	Description
from_utc	Static. Converts a `utc_time` to a `file_time`.
now	Static. Returns the current International Atomic Time.
to_utc	Static. Converts a `file_time` to `utc_time`.

Non-members

Expand table

Name	Description
from_stream	Parse a `file_clock` from the given stream using the specified format.

Name	Description
operator<<	Output `file_time` to the given stream.

Convenience type aliases

Expand table

Name	Description
`file_clock::duration`	In Microsoft's implementation, it's a synonym for `duration<long long, ratio<1, 10'000'000>`. It represents a duration of time measured in units of 100 nanoseconds.
`file_clock::time_point`	A synonym for `time_point<File_time_clock>`. Used to represent a `time_point` for this clock.
`file_clock::period`	In Microsoft's implementation, it's a synonym for `ratio<1, 10'000'000>`. It represents the time in seconds (100 nanoseconds) between each tick in the duration.
`file_clock::rep`	A synonym for the type (`long long`) used to represent the integral units in this clock's `file_clock::duration`.

Related

Expand table

Name	Description
`file_time`	A synonym for `time_point<file_clock, Duration>`. Represents a `time_point` for a file_clock. You specify the `Duration`. Defined in `std::chrono`

Public constants

Expand table

Name	Description
file_clock::is_steady constant	Indicates whether the clock type is steady. Its value is `false`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`from_utc`

Static method that converts a `utc_time` to a `file_time`.

```
C++
```

```
template <class Duration>
static file_time<common_type_t<_Duration, chrono::seconds>>
from_utc(const utc_time<Duration>& t);
```

Parameters

`t`
The `utc_time` to convert.

Return value

A `file_time` that represents the equivalent `utc_time` as *`t`*. It's calculated as `utc_clock::to_sys(utc_time).time_since_epoch()` minus the number of leap seconds before January 1, 2017 (27). Windows 10 version 1809 and Windows Server 2019 introduced support for leap seconds. That support is enabled by default, though it can be disabled. When enabled, only leap seconds after July 2018 (not the ones between January 1, 2017 and July 2018) are included in the time.

We recommend you use `std::chrono::clock_cast` to convert time points between clocks rather than call this function directly. This is particularly relevant for `file_clock` because the ISO C++ Standard allows this type to define either `to_utc` and `from_utc`, or `to_sys` and `from_sys`. Since which is implemented may vary by vendor, you can use `clock_cast` instead, which is provided by all library vendors.

Example: `from_utc`

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    std::cout << clock_cast<file_clock>(utc_clock::now());

    return 0;
}
```

Output

```
2021-08-17 00:20:41.2594557
```

`to_utc`

Static method that converts a `file_time` to a `utc_time`.

C++

```
template <class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const file_time<Duration>& t);
```

Parameters

`t`

The `file_time` to convert.

Return Value

A `utc_time` that represents the equivalent `file_time` as *`t`*. We recommend you use `std::chrono::clock_cast` to convert time points between clocks, rather than call this function directly. This is particularly relevant for `file_clock` because the ISO C++ Standard allows this type to define either `to_utc` and `from_utc`, or `to_sys` and `from_sys`. Since which is implemented may vary by vendor, you can use `clock_cast` instead, which is provided by all library vendors.

Example: `to_utc`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    std::cout << clock_cast<std::chrono::utc_clock>(file_clock::now());

    return 0;
}
```

Output

```
2021-08-17 00:20:41.2594557
```

`is_steady`

Static value that specifies whether the clock type is *steady*. In the Microsoft implementation, `is_steady` is **`false`**. This makes this clock unsuitable for measuring how long an operation takes because an unsteady clock can be adjusted while you're timing something so the measured time may be off or even negative. Use a high_resolution_clock to time events, instead.

C++

```
static const bool is_steady = false;
```

now

Static method that returns the current system time, with nanosecond resolution, adjusted by the epoch of the `file_clock`.

C++

```
static time_point now() noexcept;
```

Return Value

A time_point object that represents the current time.

See also

<chrono>
gps_clock class
high_resolution_clock
local_t struct
steady_clock struct
system_clock struct
tai_clock class
utc_clock class
Header Files Reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`gps_clock` class

Article • 06/15/2022

A clock for Global Positioning System (GPS) time.

Syntax

```
C++
```

```
class gps_clock; // C++20
```

Remarks

This clock measures time since Sunday, 6 January 1980 00:00:00 UTC.

This clock doesn't account for leap seconds. Whenever a leap second is added into UTC time, UTC falls another second behind GPS time.

GPS time is behind TAI time by 19 seconds because 10 seconds were added to TAI time in 1972 to account for the leap seconds that had accumulated to that point, and 9 more leap seconds were added to TAI time between 1970 and 1980.

Members

Name	Description
from_utc	Static. Converts a `utc_time` to a `gps_time`.
now	Static. Returns the GPS current time.
to_utc	Static. Converts a `gps_time` to a `utc_time`.

Non-members

Name	Description
from_stream	Parse a `gps_time` from the given stream using the specified format.
operator<<	Output a `gps_time` to the given stream.

Convenience type aliases

Name	Description
`gps_clock::duration`	In Microsoft's implementation, it's a synonym for `duration<long long, ratio<1, 10'000'000>`. It represents a duration of time, as measured in units of 100 nanoseconds.
`gps_clock::period`	In Microsoft's implementation, it's a synonym for `ratio<1, 10'000'000>`. It represents the time in fractions of a second (100 nanoseconds) between each clock tick in the duration.
`gps_clock::rep`	A synonym for the type (`long long`) used to represent the integral units in this clock's `gps_clock::duration`.
`gps_clock::time_point`	A synonym for `time_point<gps_clock>`. Used to represent a `time_point` for this clock.

Related

Name	Description
`gps_time`	A synonym for `template <class Duration> using gps_time = time_point<gps_clock, Duration>`. Used to represent a time_point for a `gps_clock`. You specify the `Duration`. Defined in `std::chrono`
`gps_seconds`	A synonym for `using gps_seconds = gps_time<seconds>;` A count of seconds, represented by a `time_point` that is associated with a `gps_clock`. Defined in `std::chrono`

Public Constants

Name	Description
gps_clock::is_steady constant	Indicates whether the clock type is steady. Its value is `false`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`from_utc`

Static method that converts a `utc_time` to a `gps_time`.

```
C++
```

```
template<class Duration>
static gps_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>& t) noexcept;
```

Parameters

t

The `utc_time` to convert.

Return value

A `gps_time` set to the same point in time as the `utc_time` *t*. It's computed as `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 315964809s` where `315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s`. Which is the number of seconds between the `utc_clock` epoch and the `gps_clock` epoch.

`to_utc`

Static method that converts a `gps_time` to a `utc_time`.

```
C++
```

```
template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const gps_time<Duration>& t) noexcept;
```

Parameters

t

The `gps_time` to convert.

Return Value

A `utc_time` set to the same point in time as the `gps_time`. It's computed as `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 315964809s` where `315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s`. That's the number of seconds between the epochs of the `utc_clock` and the `gps_clock`.

`is_steady`

Static value that specifies whether the clock type is *steady*. In Microsoft's implementation, `is_steady_constant` is always **`false`**. Because the `gps_clock` isn't steady, you can't reliably use this clock to take the time before an event, the time after an event, and subtract them to get the duration of the event because the clock may be adjusted during that time.

C++

```
static const bool is_steady = system_clock::is_steady; //
system_clock::is_steady equals false
```

`now`

Static method that returns the current GPS time.

C++

```
static time_point now() noexcept;
```

Return Value

A time_point object that represents the current time. The returned time point is effectively `from_utc(utc_clock::now())`.

See also

<chrono>
file_clock class
high_resolution_clock
local_t struct
steady_clock struct

system_clock struct
tai_clock class
Header Files Reference

hh_mm_ss class

Article • 06/15/2022

Splits a `std::chrono::duration` into hours:minutes:seconds, with the precision of the split determined by the `Duration` template parameter. This class is used primarily as a formatting tool.

`Duration` must be a specialization of `std::chrono::duration`.

Syntax

```
C++
```

```
template<class Duration> class hh_mm_ss; // C++20
```

Members

Name	Description
Constructor	Construct a `hh_mm_ss`.
is_negative	Indicates whether the time duration is negative.
hours	Get the hours value.
minutes	Get the minutes value.
seconds	Get the seconds value.
subseconds	Get the subseconds value.
to_duration	Return a `duration_cast<precision>`.
hh_mm_ss::operator precision	Returns the precision of this `hh_mm_ss`.

Non-Members

Name	Description
hh_mm_ss::operator<<	Output a `hh_mm_ss` to the given stream.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`hh_mm_ss::hh_mm_ss` constructor

Constructs a `hh_mm_ss`.

```
C++
```

```
constexpr explicit hh_mm_ss(Duration d);
```

Parameters

`d`

Constructs an `hh_mm_ss` represents the Duration *`d`* with precision defined as `duration<common_type_t<Duration::rep, seconds::rep>, ratio<1, 10fractional_width>>`

Remarks

If `treat_as_floating_point_v<precision::rep>` is `true`, the precision is initialized with `abs(d)-hours()-minutes()-seconds()`. Otherwise, the precision is initialized with `duration_cast<precision>(abs(d)-hours()-minutes()-seconds())`

`hh_mm_ss::is_negative`

Get whether the time duration is negative.

```
C++
```

```
constexpr bool is_negative() const noexcept;
```

Return value

`true` if the `hh_mm_ss` value is negative. Otherwise, `false`.

`hh_mm_ss::hours`

Get the hours portion of this `hh_mm_ss`.

```
C++
```

```
constexpr chrono::hours hours() const noexcept;
```

Return value

The `hours` value of this `hh_mm_ss`.

`hh_mm_ss::minutes`

Get the minutes portion of this `hh_mm_ss`.

```
C++
```

```
constexpr chrono::minutes minutes() const noexcept;
```

Return value

The `minutes` value of this `hh_mm_ss`.

`hh_mm_ss::seconds`

Get the seconds portion of this `hh_mm_ss`.

```
C++
```

```
constexpr chrono::seconds seconds() const noexcept;
```

Return value

The `seconds` value of this `hh_mm_ss`.

`hh_mm_ss::subseconds`

Get the subseconds portion of this `hh_mm_ss`.

```
C++
```

```
constexpr precision subseconds() const noexcept;
```

Return value

The `subseconds` value of this `hh_mm_ss`.
If `std::chrono::treat_as_floating_point_v<precision::rep>` is `true`, then this is calculated as `abs(d)-hours()-minutes()-seconds()`. Otherwise, it's calculated as `std::chrono::duration_cast<precision>(abs(d)-hours()-minutes()-seconds())`

`hh_mm_ss::operator precision`

```
C++
```

```
constexpr explicit operator precision() const noexcept;
```

Return value

Returns to_duration

`hh_mm_ss::to_duration`

```
C++
```

```
constexpr precision to_duration() const noexcept;
```

Return value

If `is_neg` is `true`, returns `-(h+m+s+ss)`. Otherwise, returns `h+m+s+ss`.

See also

<chrono>
Header Files Reference

`high_resolution_clock` struct

Article • 03/02/2023

Represents a *high_resolution* clock with the smallest possible tick period. Specifically, the period is a nanosecond, or `ratio<1, 1000000000>`

In Microsoft's implementation, `high_resolution_clock` is a synonym for steady_clock.

Syntax

```
C++
```

```
using high_resolution_clock = steady_clock;
```

Members

Convenience type aliases

Name	Description
`high_resolution_clock::duration`	In Microsoft's implementation, it's a synonym for `nanoseconds`, which is defined as `duration<long long, nano>`. It represents a duration of time measured in billionths of a second.
`high_resolution_clock::period`	In Microsoft's implementation, it's a synonym for `nano`, which is defined as `std::ratio<1i64, 1000000000i64>` . It represents the time in seconds (one billionth of a second) between each tick in the duration.
`high_resolution_clock::rep`	A synonym for **`long long`**, which is the type used in the Microsoft implementation to represent the number of clock ticks in a `high_resolution_clock::duration`.
`high_resolution_clock::time_point`	A synonym for `time_point<steady_clock>`. Used to represent a `time_point` for this clock.

Functions

Name	Description

Name	Description
`now`	Returns the current `high_resolution_clock` time.

Constants

Name	Description
`is_steady`	Holds **`true`**. A `high_resolution_clock` is *steady*. That means you can use this clock to take the time before an event, the time after an event, and reliably subtract them to get the duration of the event because the clock won't be adjusted during that time.

See also

<chrono>
file_clock class
gps_clock class
local_t struct
steady_clock struct
system_clock struct
tai_clock class
utc_clock class
Header Files Reference

`is_clock` structure

Article • 06/15/2022

A type trait that determines whether the specified type meets the requirements to be a clock.

Syntax

```
C++
```

```
template<class T> struct is_clock; // c++ 20
```

Helper variable template

```
C++
```

```
 template<class T> inline constexpr bool is_clock_v = is_clock<T>::value; //
c++ 20
```

Parameters

`T`

The type to test.

Members

Name	Description
value	Indicates whether `T` satisfies the requirements to be a clock.
operator ()	Returns value.
operator bool	Returns value.

Remarks

A clock has a `rep`, `period`, `duration`, `time_point`, `is_steady`, and a `now()` function.

For more details about the requirements to be a C++17 clock, see Cpp17Clock requirements.

The following code works because `is_clock`, derives from `Cpp17UnaryTypeTrait`, which derives from `integral_constant`. This is where `value_type`, which is a `bool`, and `type`, which is a `std::integral_constant<bool, value>` come from.

Example

C++

```
#include <iostream>
#include <chrono>

using namespace `std::chrono`;

int main()
{
    is_clock<system_clock> ic;

    std::cout << std::boolalpha << ic.value << ", " << ic() << ", " <<
(bool)ic;
    return 0;
}
```

Output

```
true, true, true
```

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`Value`

Get whether the specified type satisfies the requirements to be a clock.

C++

```
static constexpr T value;
```

Return value

`true` if specified type meets the requirements to be a clock. Otherwise, `false`.

`operator()`

```
C++
```

```
constexpr value_type operator()() const noexcept
```

Returns `value`, that is, whether the specified type meets the requirements to be a clock.

Return value

`true` if specified type meets the requirements to be a clock. Otherwise, `false`.

`operator bool`

```
C++
```

```
constexpr operator value_type() const noexcept
```

Returns `value`, that is, whether the specified type meets the requirements to be a clock.

Return value

`true` if specified type meets the requirements to be a clock. Otherwise, `false`.

See also

<chrono>
Header Files Reference

`last_spec` struct

Article • 06/15/2022

Indicates the last item in a month. For example, the last day of the week of a month, such as the last Tuesday of February 2020. Or the last day of a month, such as the last day of April 2019.

Syntax

C++

```
struct last_spec; // C++20
inline constexpr last_spec last{} // C++20
```

Example: `last`

Use `last_spec` to indicate the last item in a month. The following example uses `last` to indicate the:

- Last day in December as a `monthDayLast`
- Last Friday in July of 2021 as a `year_month_weekday_last`
- Last day in April of 1975 as a `year_month_day_last`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    constexpr auto monthDayLast{December/last}; // 'last' resolves to
last_spec
    std::cout << monthDayLast << '\n';

    constexpr auto yearMonthWeekDayLast{year(2021)/July/Friday[last]}; //
'last' resolves to last_spec
    std::cout << yearMonthWeekDayLast << '\n';

    constexpr auto yearMonthDayLast{ April / last / 1975 };
    std::cout << yearMonthDayLast << "\n";
```

```
    return 0;
}
```

Output

```
Dec/last
2021/Jul/Fri[last]
1975/Apr/last
```

Members

Name	Description
Constructor	Construct a `last_spec`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Construct a `last_spec`.

C++

```
explicit last_spec() = default;
```

Remarks

You typically won't create a `last_spec` instance directly. You'll use `last` as shown above in Example last.

See also

<chrono>
month_day_last Class

`leap_second` class

Article • 06/06/2022

A `leap_second` represents a date and a value for an inserted leap second. The library creates `leap_second` objects as part of its time zone database initialization. It provides `const` access to the created objects. You can't construct or copy a `leap_second` object, and use of the default move constructor or default move assignment operator results in undefined behavior.

Syntax

```
C++

class leap_second  // Since C++20
```

Members

Name	Description
Constructor	Construct a `leap_second`
date	Gets the date and time when the leap second was inserted.
value	Gets the positive or negative number of leap seconds inserted.

Requirements

Header: `<chrono>`

Microsoft C++ supports the `leap_second` class starting in Visual Studio 2019 version 16.10. The `leap_second` class is a C++20 feature. The /std:c++latest compiler option is required.

An international standards body specifies when new leap seconds should be accounted for. A database of these changes was added starting in Windows 10 version 1809 and later, and Windows Server 2019 and later.

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Construct a `leap_second`.

```
C++
```

```
leap_second(const leap_second& ls) = default;
```

Parameters

`ls`
The `leap_second` that this object will be a copy of.

Remarks

You can't construct `leap_second` objects except by copying an existing `leap_second`. `leap_second` instances are provided by the library. You typically don't create them.

`date`

Gets the date and time when the leap second was inserted.

```
C++
```

```
constexpr sys_seconds date() const noexcept;  // Since C++20
```

Return value

Returns a `date` containing the date and time when the leap second was inserted.

Remarks

Microsoft C++ supports `leap_second::date` starting in Visual Studio 2019 version 16.10. The function is a C++20 feature that requires the /std:c++latest compiler option.

`name`

Gets the name of this `leap_second`.

```
```

C++

```
constexpr seconds value() const noexcept;  // Since C++20
```

Return value

Returns `+1s` or `-1s` to indicate whether a positive or negative leap second was inserted.

Remarks

Microsoft C++ supports `leap_second::value` starting in Visual Studio 2019 version 16.10. The function is a C++20 feature that requires the /std:c++latest compiler option.

See also

<chrono>
get_leap_second_info function
leap_second_info struct
Header files reference

`leap_second_info` struct

Article • 06/06/2022

Call get_leap_second_info to get a `leap_second_info`. This type only has public data members. It doesn't have base classes or members other than the ones specified.

Syntax

```
C++

struct leap_second_info; // C++20
```

Public members

Name	Description
`is_leap_second`	The `is_leap_second` member is **`true`** only if the specified time point occurs during the insertion of a positive leap second.
`elapsed`	The `elapsed` member holds the sum of all the leap seconds between the epoch date (the starting date from which the clock measures time) `1970-01-01` and the specified time. If `is_leap_second` is **`true`**, the leap second referred to by the specified time is included in the `elapsed` sum.

Requirements

Header: `<chrono>`

Microsoft C++ supports the `leap_second` class starting in Visual Studio 2019 version 16.10. The `leap_second` class is a C++20 feature. The /std:c++latest compiler option is required.

An international standards body specifies when new leap seconds should be accounted for. A database of these changes is available for Windows 10 version 1809 and later, and Windows Server 2019 and later.

Namespace: `std::chrono`

See also

<chrono>
get_leap_second_info
leap_second class
Header files reference

`local_info` struct

Article • 10/18/2021

Provides a low-level interface to time zone information about the result of converting a `local_time` to a `sys_time`.

Syntax

```
C++

struct local_info;
```

Members

Function	Description
result	A code describing the result of the conversion.
first, second	The result of the conversion.

Member constants

Name	Value	Description
`unique`	0	The result of the conversion is unique.
`nonexistent`	1	There isn't a corresponding `sys_time` to convert the `local_time` to.
`ambiguous`	2	The result of the conversion is ambiguous.

Non-members

Function	Description
operator<<	Output a `local_info` to a stream.

`result`

The result of converting a `local_time` to a `sys_time`.

```
C++
```

```
int result;
```

Return value

The result can be one of the following values:

- `unique`: The `local_time` was successfully converted to a `sys_time`.
- `nonexistent`: There isn't a mapping from the `local_time` to the `sys_time`.
- `ambiguous`: The `local_time` lands during a transition between daylight saving time and standard time and so could be converted to two `sys_time` values.

Remarks

For information about how `first` and `second` are set based on `result`, see the following section about first, second.

`first`, `second`

Provides the result of converting a `local_time` to a `sys_time`.

```
C++
```

```
sys_seconds begin;
sys_seconds end;
```

Return value

`first` and `second` are set based on `result` as follows:

`result`	`first`	`second`
`unique`	Contains the value of `local_time` converted to `sys_time`	zero-initialized
`nonexistent`	A `sys_info` that ends just before the `local_time`	A `sys_info` that begins just after the `local_time`.
`ambiguous`	A `sys_info` that ends just after the `local_time`	A `sys_info` that starts just before the `local_time`.

`ambiguous` result code

During the transition from daylight saving time to standard time in the fall, clocks essentially gain an extra hour. This can be confusing because doesn't the transition to standard time mean losing an hour? By falling back an hour, the hour before the transition will be repeated after the clock adjusts for standard time. Consider the change to standard time in New York, which happens on the first Sunday in November at 2:00am. First, 1:00am goes by. At 2am, the clock transitions to standard time, so now it's 1:00am again. That means the time between 1am and 2am will be "repeated", effectively adding an hour. See ambiguous_local_time for an example.

If a `local_time` specifies a time during this "extra" hour, it isn't clear how to convert it. Should the converted time be treated as the "first" time that hour happens, or the "second"? If the enum choose isn't specified to indicate which it should be, expect `result` to be set to `ambiguous`, and `first` and `second` will reflect the two choices for what the time the `local_time` could be converted to.

`nonexistent` result code

During the transition from standard time to daylight saving time in the spring, clocks essentially lose an hour. This can be confusing because doesn't the transition to daylight saving time mean adding an hour? By "springing forward" an hour, the hour following the transition is effectively removed. Consider the change to daylight saving time in New York, which happens on the second Sunday in March at 2am. At 2am, the clock transitions to daylight savings time and now reads 3:00am. If the `local_time` being converted is 2:30am, for example, that time is during the period that was "removed" and so you can expect `result` to have the `nonexistent` code. See nonexistent_local_time for an example.

Example: get a `local_info`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    const auto& timeZoneDatabase = get_tzdb();
    const auto& currentZone = timeZoneDatabase.current_zone();
```

```
    local_time<system_clock::duration> lt = currentZone-
>to_local(system_clock::now());

    auto localInfo = currentZone->get_info(lt);

    std::cout << "local_time: " << lt << "\n";
    std::cout << localInfo << "\n";

    return 0;
}
```

Output

```
local_time: 2021-09-08 15:37:57.6853963
result: unique, first: (begin: 2021-03-14 10:00:00, end: 2021-11-07
09:00:00, offset: -25200s, save: 60min, abbrev: PDT)
```

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

<chrono>
ambiguous_local_time exception
choose enum
file_clock class
high_resolution_clock
local_t struct
non_existent exception
system_clock struct
tai_clock class
utc_clock class
Header Files Reference

`local_t` struct

Article • 06/15/2022

A pseudo-clock.

Syntax

```
C++
```

```
struct local_t {}; // C++20
```

Remarks

Use this struct as an argument to the `time_point` template to indicate that a `time_point` represents local time. Specifying `local_t` doesn't specify the time zone of the local time.

As an example of how `local_t` is used, consider the declaration `local_days ld{Sunday[1] / January / 2021};` This ultimately resolves to `time_point<local_t, duration<long long>;` where `local_t` indicates that the time_point represents local time.

Non-members

Name	Description
from_stream	Parse a `local_time` from the given stream using the specified format.
operator<<	Output a `local_time` to the given stream.

Related typedefs

Name	Description
`local_days`	A synonym for l`local_time<days>`. Defined in `std::chrono`.
`local_seconds`	A synonym for `local_time<seconds>`. Defined in `std::chrono`.
`local_time`	A synonym for `template <class Duration> using local_time = time_point<local_t, Duration>`. Useful for representing a `time_point` for a local time. You specify the `Duration`. Defined in `std::chrono`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

See also

<chrono>
file_clock class
gps_clock class
high_resolution_clock
steady_clock struct
system_clock struct
tai_clock class
Header Files Reference

`month` class

Article • 10/13/2022

Represents a month of a year. For example, July.

Syntax

```
C++
```

```
class month; // C++ 20
```

Remarks

A `month` normally holds values in the range [1, 12]. It can also hold non-negative values outside this range. See Month constants, below, for constants that you can use with the `month` class.

Members

Name	Description
Constructors	Construct a `month`.
ok	Verify that the month value is in the valid range [1,12].
operator++	Increment the `month`.
operator+=	Add the specified number of months to this `month`.
operator--	Decrement this `month`.
operator-=	Subtract the specified number of months from this `month`.
operator unsigned	Get the `month` value.

Non-members

Name	Description
from_stream	Parse a `month` from the given stream using the specified format.

Name	Description
operator+	Add specified number of months to this `month`, returning a new `month` instance.
operator-	Subtract the specified number of months from this month. Returns a new `month` instance.
operator==	Determine whether two months are equal.
operator<=>	Compare this month against another month. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `month` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `month`.

```
C++

1) month() = default;
2) explicit constexpr month(unsigned m) noexcept;
```

Parameters

m

Construct a `month` with value *m*.

Remarks

1) The default constructor doesn't initialize the day value.
2) Construct a `month` with the day value initialized to `m`.

ok

Check if the value stored in this `month` is in the valid range.

```
C++

constexpr bool ok() const noexcept;
```

Return value

`true` if the month value is in the range [1,12]. Otherwise, `false`.

`operator++`

Increment the month value.

```
C++

1) constexpr month& operator++() noexcept;
2) constexpr month operator++(int) noexcept;
```

Return value

1) A reference to `*this` month *after* it has been incremented (a postfix increment).

2) A copy of the `month`, *before* it has been incremented (a prefix increment).

Example: `operator++`

```
C++

// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    month m{ January };
    month m2{4}; // April

    std::cout << m << " " << ++m << "\n"; // constexpr month& operator++()
noexcept
    std::cout << m << " " << m++ << "\n"; // constexpr month operator++(int)
noexcept
    std::cout << m << "\n";
```

```
    std::cout << m2 << "\n";

    return 0;
}
```

Output

```
Jan Feb
Feb Feb
Mar
Apr
```

Remarks

If the result exceeds 12, it's set to 1.

`operator--`

Subtract 1 from the month value.

C++

```
1) constexpr month& operator--() noexcept;
2) constexpr month operator--(int) noexcept;
```

Return value

1) A reference to `*this` `month` *after* it has been decremented (a postfix decrement).
2) A copy of the `month` *before* it has been decremented (a prefix decrement).

Example: `operator--`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
    month m{May};
```

```
    cout << m << " " << --m << "\n"; // constexpr month& operator++()
noexcept
    cout << m << " " << m-- << "\n"; // constexpr month operator++(int)
noexcept
    cout << m << "\n";

    return 0;
}
```

Output

```
May Apr
Apr Apr
Mar
```

Remarks

If the decremented result is less than 1, it's set to 12.

`operator+=`

Add months to this `month`.

C++

```
constexpr month& operator+=(const months& m) noexcept;
```

Parameters

m

The number of months to add.

Return value

`*this`

`operator-=`

Subtract `months` from this `month`.

C++

```
constexpr month& operator-=(const months& m) noexcept;
```

Parameters

m

The months to subtract.

Return value

`*this`.

`operator unsigned`

Get the unsigned `month` value.

C++

```
explicit constexpr operator unsigned() const noexcept;
```

Return value

The unsigned value of this `month`

Example: `operator unsigned()`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    month m{July};
    unsigned monthValue = static_cast<unsigned>(m);
    std::cout << monthValue << "\n";

    return 0;
}
```

Output

```
7
```

Month constants

(C++20) The `<chrono>` header defines the following constants that you can use with `month` for greater convenience, type-safety, and maintainability of your code. These constants are in scope when `std::chrono` is in scope.

C++

```
// Calendrical constants
inline constexpr month January{1};
inline constexpr month February{2};
inline constexpr month March{3};
inline constexpr month April{4};
inline constexpr month May{5};
inline constexpr month June{6};
inline constexpr month July{7};
inline constexpr month August{8};
inline constexpr month September{9};
inline constexpr month October{10};
inline constexpr month November{11};
inline constexpr month December{12};
```

See also

<chrono>
month_day Class
month_day_last Class
month_weekday Class
month_weekday_last class

`month_day` class

Article • 06/15/2022

Represents a specific day of a specific month. The year isn't specified.

Syntax

```
C++
```

```
class month_day; // C++20
```

Members

Name	Description
Constructors	Construct a `month_day`.
day	Return the day value.
month	Return the month value.
ok	Check if the `month_day` is valid.

Non-members

Name	Description
from_stream	Parse a `month_day` from the given stream using the specified format.
operator==	Determine whether two months are equal.
operator<=>	Compare this month against another month. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `month_day` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `month_day`.

```
C++
```

```
1) month_day() = default;
2) constexpr month_day(const month& m, const day& d) noexcept;
```

Parameters

d

Construct a `month_day` with a day value of *d*.

m

Construct a `month_day` with a month value of *m*.

Remarks

1) The default constructor doesn't initialize the month or day values.
2) Construct a `month_day` with the month value initialized to `m` and the day value initialized to `d`.

For information about C++ 20 syntax to specify dates, see operator/

Example: Create a `month_day`

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    month_day md(30d/July);
    month_day md2 = July/30; // another way to construct a month_day

    std::cout << md << '\n' << md2;
```

```
    return 0;
}
```

Output

```
Jul/30
Jul/30
```

day

Return the day value.

C++

```
constexpr day day() const noexcept;
```

Return value

The day value.

month

Return the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

ok

Check if the value stored in this `month_day` is valid.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the `month_day` value is valid. Otherwise, `false`.
A `month_day` is valid if the month is valid and the day is less than or equal to the number of days in that month. February has 29 days.

See also

<chrono>
month class
month_day class
month_day_last class
month_weekday class
month_weekday_last class
operator/

`month_day_last` class

Article • 06/15/2022

Represents the last day of a month.

Syntax

```
C++
```

```
class month_day_last; // C++ 20
```

Members

Name	Description
Constructor	Construct a `month_day_last`
month	Return the month value.
ok	Check if this `month_day` is valid.

Non-members

Name	Description
operator==	Determine whether two `month_days` are equal.
operator<=>	Compare this `month_day` against another `month_day`. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `month_day` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Construct a `month_day_last` instance.

```
C++
```

```
constexpr explicit month_day_last(const month& m) noexcept;
```

Parameters

`m`

Construct a `month_day_last` for the month specified by *`m`*.

Remarks

For information about C++ 20 syntax used to specify dates, see operator/

Example: Create a `month_day_last`

```
C++
```

```
// compile using: /std:c++latest
#include <chrono>

using namespace std::chrono;

int main()
{
    month_day_last mdl{ October / last };
    std::cout << mdl;

    return 0;
}
```

```
Output
```

```
Oct/last
```

`month`

Return the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

ok

Check if the month stored in this `month_day_last` is valid.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the month value in this `month_day_last` is valid. Otherwise, `false`.

See also

<chrono>
month class
month_day class
month_day_last class
month_weekday class
month_weekday_last class
operator/ Header Files Reference

`month_weekday` class

Article • 01/04/2024

Represents the nth weekday of a specific month.

Syntax

```
C++
```

```
class month_weekday; // C++20
```

Remarks

The year is unspecified.
`month_weekday` is a trivially copyable and standard-layout class type.

Members

Expand table

Name	Description
Constructor	Construct a `month_weekday` with the specified month and weekday.
month	Return the month value.
ok	Check if the `month_weekday` is valid.
weekday_indexed	Return the weekday index.

Non-members

Expand table

Name	Description
operator==	Determine whether two months are equal.
operator<<	Output a `month_weekday` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `month_weekday`. The month and weekday aren't initialized.

C++

```
constexpr month_weekday(const month& m, const weekday_indexed& wdi)
noexcept;
```

Parameters

m

Construct a `month_weekday` with a month value of *m*.

wdi

Construct a `month_weekday` with a weekday value of *wdi*.

Remarks: Constructor

For information about C++20 syntax to specify dates, see operator/

Example: Create a `month_weekday`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    month_weekday mw{ July/Monday[1] };
    std::cout << mw << '\n';
```

```
    // Another way to create a month_weekday
    month_weekday mw2 = February / Tuesday[3];
    std::cout << mw2;

    return 0;
}
```

Output

```
Jul/Mon[1]
Feb/Tue[3]
```

`month`

Get the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`ok`

Check if the value stored in this `month_weekday` is valid.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the `month_weekday` value is valid. Otherwise, `false`.
A `month_weekday` is valid if both the `month` is valid and the `weekday_indexed` value is valid.

`weekday_indexed`

Return the weekday of the month value.

```
C++
```

```
constexpr weekday_indexed weekday_indexed() const noexcept;
```

Return value

The weekday of the month.

See also

<chrono>
month class
month_day Class
month_day_last Class
month_weekday_last class

`month_weekday_last` class

Article • 06/15/2022

Represents the last weekday of a month.

Syntax

```
C++
```

```
class month_weekday_last; // C++20
```

Remarks

The year is unspecified.
`month_weekday_last` is a trivially copyable and standard-layout class type.

Members

Name	Description
Constructors	Constructs a `month_weekday_last`
month	Get the month value.
ok	Check if the `month_weekday_last` is valid.
weekday_last	Get the weekday value.

Non-members

Name	Description
operator==	Determine whether two `month_weekday_last` instances are equal.
operator<<	Output a `month_weekday_last` to the specified stream.

Requirements

Header: `<chrono>` Since C++ 20

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Construct a `month_weekday_last` initialized with a `month` and `weekday_last` value.

```
C++
```

```
constexpr month_weekday_last(const month& m, const weekday_last& wdl)
noexcept;
```

Parameters

`m` The `month` value for the created `month_weekday_last` class.

`wdl`
The `weekday_last` value for the created `month_weekday_last` class.

Remarks: Constructor

For information about C++ 20 syntax to specify dates, see operator/

Example: Create a `month_weekday_last`

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    constexpr auto mwdl{ January / Monday[last] }; // wdl is the last Monday
of January of an unspecified year
    std::cout << mwdl << "\n";

    const auto theMonth = August;
    const auto wdl = Friday[last];
    month_weekday_last mwdl2(theMonth, wdl);
    std::cout << mwdl2;
```

```
    return 0;
}
```

Output

```
Jan/Mon[last]
Aug/Fri[last]
```

`month()`

Return the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`ok`

Check if the value stored in this `month_weekday_last` is in the valid range.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the `month` and `weekday_last` value is in the valid range. Otherwise, `false`.

`weekday_last`

Get the `weekday_last` value.

C++

```
constexpr weekday_last weekday_last() const noexcept;
```

Return value

The `weekday_last` value.

See also

<chrono>
month class
month_day class
month_day_last class
month_weekday class
operator/

`nonexistent_local_time` class

Article • 06/15/2022

This exception is thrown when attempting to convert a `local_time` to a non-existent `sys_time`.

Syntax

C++

```
class nonexistent_local_time : public runtime_error; // c++ 20
```

Remarks

During the transition from standard time to daylight saving time in the spring, clocks essentially lose an hour. This can be confusing because doesn't the transition to daylight saving time mean adding an hour? By "springing forward" an hour, the hour following the transition is effectively removed. Consider the change to daylight saving time in New York, which happens on the second Sunday in March at 2am. At 2am, the clock transitions to daylight savings time and now reads 3:00am. If the `local_time` being converted is 2:30am, for example, that time is during the period that was "removed" and so is "nonexistent".

The following example demonstrates a nonexistent time conversion error. In this example, daylight saving time in New York starts at 2:00am. The time being converted is 2:30am. That's during the hour that is removed because of the transition from standard time to daylight saving time. So, a `nonexistent_local_time` exception is thrown.

Example: `nonexistent_local_time`

C++

```
#include <chrono>
#include <iostream>

using namespace std::chrono;

int main()
{
    try
    {
```

```
        // The following will throw an exception because the local time
being converted is during
        // the hour that is "removed" when the clock advances an hour for
daylight saving time.
        auto zt = zoned_time{"America/New_York",
local_days{Sunday[2]/March/2016} + 2h + 30min};
    } catch (const nonexistent_local_time& e)
    {
        std::cout << e.what() << '\n';
    }
    return 0;
}
```

Output

```
2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC
```

Members

Name	Description
Constructors	Construct a `nonexistent_local_time`.
what	Gets a string describing the non-existent time.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `nonexistent_local_time`.

C++

```
template<class Duration>
nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);
```

Parameters

tp

The local_time that failed conversion.

i

Information about the attempted conversion. See local_info for details.

Remarks

You typically won't create this exception. It's thrown by functions that convert a `local_time` to a `sys_time`.

`what`

Gets a string describing why the time is non-existent.

C++

```
[nodiscard] virtual const char* what() const noexcept;
```

Return value

A string describing why the time is non-existent. For example:

Output

```
2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC
```

See also

<chrono>
to_sys
ambiguous_local_time
Header Files Reference

`steady_clock` struct

Article • 06/15/2022

Represents a clock that is never adjusted and that moves forward at a uniform rate.

Syntax

C++

```
struct steady_clock;
```

Remarks

A `steady_clock` is a monotonic clock, which means that the time it reports only moves forward. The value returned by a call to `now` is always less than or equal to the value that is returned by the next call to `now`.

A `steady_clock` ensures that the time between ticks is constant, making it better for measuring intervals than using a system clock. The system clock provides wall-clock time. The problem with using wall-clock time to measure elapsed time is that wall-clock time may get modified while measuring a time interval. It could be modified by synchronizing with another clock over the network, by a transition to daylight savings time, and so on. A `steady_clock` isn't subject to these adjustments, making it the preferred way to keep track of elapsed time.

`high_resolution_clock` is a typedef for `steady_clock`. On Windows, `steady_clock` wraps the `QueryPerformanceCounter` function.

Example

C++

```
#include <chrono>
#include <thread>
#include <iostream>

using namespace std::chrono;

int main()
{
    auto begin = steady_clock::now();
```

```
    std::this_thread::sleep_for(std::chrono::seconds(1));
    auto end = steady_clock::now();

    std::cout << "Elapsed seconds:      "
        << duration_cast<seconds>(end - begin).count()
        << "s\n";

    std::cout << "Elapsed milliseconds: "
        << duration_cast<milliseconds>(end - begin).count()
        << "ms\n";

    std::cout << "Elapsed microseconds: "
        << duration_cast<microseconds>(end - begin).count()
        << "us\n";

    std::cout << "Elapsed nanoseconds:  "
        << duration_cast<nanoseconds>(end - begin).count()
        << " ns\n";

    return 0;
}
```

Output

```
Elapsed seconds:      1s
Elapsed milliseconds: 1007ms
Elapsed microseconds: 1007266us
Elapsed nanoseconds:  1007266700 ns
```

Convenience type aliases

Name	Description
`steady_clock::duration`	In Microsoft's implementation, it's a synonym for `nanoseconds`, which is defined as `duration<long long, nano>`. It represents a duration of time measured in billionths of a second.
`steady_clock::period`	In Microsoft's implementation, it's a synonym for `nano`, which is defined as `std::ratio<1i64, 1000000000i64>` . It represents the time in seconds (one billionth of a second) between each tick in the duration.
`steady_clock::rep`	A synonym for `long long`, which is the type used in the Microsoft implementation to represent the number of clock ticks in a `steady_clock::duration`
`steady_clock::time_point`	A synonym for `time_point<steady_clock>`. Used to represent a `time_point` for this clock.

Public functions

Function	Description
`now`	Returns the current time from the `steady_clock` as a `time_point` value.

Public constants

Name	Description
`steady_clock::is_steady`	Holds **`true`**. A `steady_clock` is *steady*. That means you can use this clock to get the time before an event, the time after an event, and reliably subtract them to get the duration of the event because the clock won't be adjusted during that time.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

See also

<chrono>
file_clock class
high_resolution_clock
local_t struct
system_clock struct
tai_clock class
utc_clock class
Header Files Reference

`sys_info` struct

Article • 06/15/2022

Provides a low-level interface to time zone information.

Syntax

```
C++
```

```
struct sys_info;
```

Remarks

Provides a low-level interface to time zone information about the result of converting a `sys_time` to a `local_time`.

Members

Function	Description
abbrev	The abbreviation used for the associated `time_zone` and `time_point.`
begin, end	The range that the `offset` and `abbrev` apply to for the associated time zone.
offset	The Universal Time Coordinated (UTC) offset in effect for the associated `time_zone` and `time_point.`
save	Daylight savings time adjustment offset.

Non-members

Function	Description
operator<<	Output a `sys_info` to a stream.

`abbrev`

The abbreviation used for the associated `time_zone` and `time_point`.

```
C++
```

```
string abbrev;
```

Return value

The abbreviation.

Remarks

Abbreviations aren't unique among the `time_zone`s. This means you can't reliably map abbreviations back to a `time_zone` and UTC offset.

`begin`, `end`

Provides the range over the associated time zone, `[begin, end)`, that the `offset` and `abbrev` apply to.

```
C++
```

```
sys_seconds begin;
sys_seconds end;
```

Return value

The returned `sys_seconds` are offsets relative to the associated `time_point`.

`offset`

The UTC offset in effect for the associated time zone and `time_point`.

```
C++
```

```
seconds offset;
```

Return value

The relationship between `local_time` and `sys_time` is: `offset = local_time - sys_time`.

`save`

Indicates whether the `sys_info` is on daylight savings time, and if so, suggests the offset this time zone might use if it weren't on daylight savings time.

C++

```
minutes save;
```

Return value

If `save != 0min`, the time zone is on daylight saving time at the associated `time_point`. In that case, `offset-save` suggests what offset this `time_zone` might use if it not on daylight saving time.

This offset isn't authoritative. To get an authoritative offset, compare with a query against time zone using a `time_point` that doesn't fall during daylight savings time. That is, compare against a `time_point` that results in a `save` value of zero.

Example: get a `sys_info`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    sys_time<system_clock::duration> st = system_clock::now();

    const auto& timeZoneDatabase = get_tzdb();
    const auto& currentZone = timeZoneDatabase.current_zone();
    auto sysInfo = currentZone->get_info(st);

    std::cout << sysInfo << "\n";

    return 0;
}
```

Output

```
begin: 2021-03-14 10:00:00, end: 2021-11-07 09:00:00, offset: -25200s, save:
60min, abbrev: PDT
```

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

<chrono>
file_clock class
high_resolution_clock
local_t struct
system_clock struct
tai_clock class
utc_clock class
Header Files Reference

`system_clock` structure

Article • 06/15/2022

A clock based on the system's real-time clock.

Syntax

```
C++
```

```
struct system_clock;
```

Remarks

This clock represents the system-wide wall clock.

The `system_clock` isn't *monotonic* or *steady*.

A clock is *monotonic* if the value that is returned by a first call to `now()` is always less than or equal to the value that is returned by a later call to `now()`. The `system_clock` isn't monotonic because the system time may be adjusted between calls to `now()`.

A clock is *steady* if it is *monotonic* and if the time between clock ticks is constant. Because the `system_clock` isn't monotonic, it isn't steady, although the time between clock ticks is constant. You can only use a steady clock to take the time before an event, the time after an event, and reliably subtract them to get the duration of the event because a steady clock won't be adjusted during that time.

Members

Convenience type aliases

Name	Description
`system_clock::duration`	In Microsoft's implementation, it's a synonym for `duration<long long, ratio<1, 10'000'000>`. It represents a duration of time measured in units of 100 nanoseconds.

Name	Description
`system_clock::period`	In Microsoft's implementation, it's a synonym for `ratio<1, 10'000'000>`. Represents the time in fractions of a second (100 nanoseconds) between each tick in the duration.
`system_clock::rep`	A synonym for `long long`. It's the type used to represent the integral units in this clock's `system_clock::duration`.
`system_clock::time_point`	A synonym for `time_point<system_clock>`. Represents a time_point for this clock.

Related

Name	Description
`sys_days`	A synonym for `using sys_days = sys_time<days>`. A count of days, represented by a `time_point` that is associated with a system_clock. Defined in `std::chrono`
`sys_seconds`	A synonym for `using sys_seconds = sys_time<seconds>`. A count of seconds, represented by a `time_point` that is associated with a system_clock. Defined in `std::chrono`
`sys_time`	A synonym for `template <class Duration> using sys_time = time_point<system_clock, Duration>`. Represents a time_point for a `system_clock`. You specify the duration. Defined in `std::chrono`.

Public methods

Name	Description
from_time_t	Static. Returns a `time_point` for a `system_clock` that most closely approximates the specified `time_t`.
now	Static. Returns the current time.
to_time_t	Static. Returns a `time_t` object that most closely approximates the specified `time_point`.

Public constants

Name	Description
system_clock::is_steady	Indicates whether the clock type is steady. It's `false`

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

`from_time_t`

Static method that returns a time_point that most closely approximates the time that is represented by *Tm*.

```
C++
```

```
static time_point from_time_t(time_t Tm) noexcept;
```

Parameters

`Tm`
A time_t object.

`is_steady`

A static value that specifies whether the clock type is *steady*. Because the `system_clock` isn't steady, you can't use this clock to take the time before an event, the time after an event, and reliably subtract them to get the duration of the event because the clock may be adjusted during the timing interval.

```
C++
```

```
static const bool is_steady = false;
```

Return Value

In this implementation, `system_clock::is_steady` always returns **`false`**.

Remarks

A clock is *steady* if it is monotonic and if the time between clock ticks is constant. The `system_clock` isn't monotonic because the value that is returned by a first call to `now()`

isn't always less than or equal to the value that is returned by a later call to `now()` because the system time can be adjusted without notice.

`now`

Static method that returns the system's current wall-clock time.

```
C++
```

```
static time_point now() noexcept;
```

Return Value

A time_point that represents the current time.

`to_time_t`

Static method that returns a time_t that most closely approximates the time that is represented by *Time*.

```
C++
```

```
static time_t to_time_t(const time_point& Time) noexcept;
```

Parameters

`Time`
A time_point object.

See also

<chrono>
file_clock class
high_resolution_clock
steady_clock struct
tai_clock class
utc_clock class
Header Files Reference

`tai_clock` class

Article • 06/15/2022

This clock reports international atomic time since 00:00:00 on Thursday, January 1, 1958. This clock doesn't account for leap seconds.

Syntax

```
C++
```

```
class tai_clock; // C++20
```

Remarks

This clock reports International Atomic Time (TAI, from the French *temps atomique international*). International Atomic Time uses a weighted average of many atomic clocks to track time.

Time kept by the TAI clock is different from UTC time. The difference is that a leap second is occasionally added to UTC time to keep the difference between UTC time and UT1 (solar time) within +- 0.9 seconds of each other. A discrepancy gradually accrues between the time kept by an atomic clock and the time kept by measuring the rotation of the earth. A discrepancy accrues because the earth's rotation speed is irregular. It slows down over time by about one second every 1.5 years, thought it also sometimes speeds up. TAI time doesn't track this discrepancy. As of this writing, TAI time is 37 seconds ahead of UTC time. That's because of an initial difference of 10 seconds at the start of 1972, plus the leap seconds that have been inserted since 1972.

The clock's epoch, or the date and time from which it starts measuring time, is `1958-01-01 00:00:00`.

Members

Name	Description
from_utc	Static. Converts a `utc_time` to a `tai_time`.
now	Static. Returns the current International Atomic Time.
to_utc	Static. Converts `tai_time` to `utc_time`.

Non-members

Name	Description
from_stream	Parse a `tai_clock` from the given stream using the specified format.
get_leap_second_info	Get information about whether the supplied time specifies a time when a leap second was inserted, and the sum of all the leap seconds between January 1, 1970 and the specified duration.
operator<<	Output a `tai_time` to the given stream.

Convenience type aliases

Name	Description
`tai_clock::duration`	In Microsoft's implementation, it's a synonym for `duration<long long, ratio<1, 10'000'000>`. It represents a duration of time measured in units of 100 nanoseconds.
`tai_clock::period`	In Microsoft's implementation, it's a synonym for `ratio<1, 10'000'000>`. It represents the time in seconds (100 nanoseconds) between each tick in the duration.
`tai_clock::rep`	A synonym for the type (`long long`) used to represent the integral units in this clock's `tai_clock::duration`.
`tai_clock::time_point`	A synonym for `time_point<tai_clock>`. Used to represent a `time_point` for this clock.

Related

Name	Description
`tai_time`	A synonym for `template <class Duration> using tai_time = time_point<tai_clock, Duration>`. Represents a time_point for a `tai_clock`. You specify the `Duration`. Defined in `std::chrono`
`tai_seconds`	A synonym for `using tai_seconds = tai_time<seconds>` A count of seconds, represented by a `time_point` that is associated with a tai_clock. Defined in `std::chrono`

Public constants

Name	Description

Name	Description
tai_clock::is_steady constant	Indicates whether the clock type is steady. Its value is `false`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`from_utc`

Static method that converts a `utc_time` to a `tai_time`.

```
C++
```

```
template <class Duration>
static tai_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>& t) noexcept;
```

Parameters

t
The `utc_time` to convert.

Return value

A `tai_time` that represents the equivalent `utc_time` as *t*. It's calculated by taking the time since the epoch of the UTC clock and adding `378691210s` where `378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s`

`to_utc`

Static method that converts a `tai_time` to a `utc_time`.

```
C++
```

```
template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
```

```
to_utc(const tai_time<Duration>& t) noexcept;
```

Parameters

`t`
The `tai_time` to convert.

Return Value

A `utc_time` that represents the equivalent `tai_time` as `t`. It's calculated as `utc_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 378691210s` where `378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s`

`is_steady`

Static value that specifies whether the clock type is *steady*. In the Microsoft implementation, `is_steady_constant` is **`false`**. Because `tai_clock` isn't steady, you can't use this clock to take the time before an event, the time after an event, and subtract them to get the duration of the event because the clock may be adjusted during that time.

C++

```
static const bool is_steady = false;
```

`now`

Static method that returns the current TAI time.

C++

```
static time_point now() noexcept;
```

Return Value

A time_point object that represents the current time. The returned time point is effectively `from_utc(utc_clock::now())`.

See also

<chrono>
file_clock class
high_resolution_clock
local_t struct
steady_clock struct
system_clock struct
utc_clock class
Header Files Reference

`time_point` class

Article • 06/15/2022

A `time_point` represents a point in time that is relative to the epoch of a clock.

Syntax

```
C++
```

```
template <class Clock, class Duration = typename Clock::duration>
class time_point;
```

Remarks

The clock that supplies the reference point for the epoch is specified by the template argument `Clock`.

Members

Convenience type aliases

Name	Description
`clock`	Synonym for the template parameter `Clock`.
`duration`	Synonym for the template parameter `Duration`.
`period`	Synonym for the nested type name `duration::period`.
`rep`	Synonym for the nested type name `duration::rep`.

Public constructors

Name	Description
time_point	Constructs a `time_point` object.

Public methods

Name	Description
max	Specifies the upper limit for `rep`.
min	Specifies the lower limit for `rep`.
time_since_epoch	Returns the amount of time between this `time_point` and the clock's epoch (or time and date that the clock starts measuring time).

Public operators

Name	Description
operator+=	Adds a specified value to the stored duration.
operator-=	Subtracts a specified value from the stored duration.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

`max`

Static method that returns the upper bound for values of type `rep`.

```
C++
```

```
static constexpr time_point max();
```

Return value

In effect, returns `time_point(duration::max())`.

`min`

Static method that returns the lower bound for values of type `rep`.

```
C++
```

```
static constexpr time_point min();
```

Return value

In effect, returns `time_point(duration::min())`.

`operator+=`

Adds a specified value to the stored duration value.

```
C++
```

```
time_point& operator+=(const duration& Dur);
```

Parameters

`Dur`
A `duration` object.

Return value

The `time_point` object after the addition is done.

`operator-=`

Subtracts a specified value from the stored duration value.

```
C++
```

```
time_point& operator-=(const duration& Dur);
```

Parameters

`Dur`
A `duration` object.

Return value

The `time_point` object after the subtraction is done.

`time_point` constructor

Constructs a `time_point` object.

```
C++
```

```
constexpr time_point();

constexpr explicit time_point(const duration& Dur);

template <class Duration2>
constexpr time_point(const time_point<clock, Duration2>& Tp);
```

Parameters

`Dur`
A duration object.

`Tp`
A `time_point` object.

Remarks

The first constructor constructs an object whose stored `duration` value is equal to duration::zero.

The second constructor constructs an object whose stored duration value is equal to *`Dur`*. Unless `is_convertible<Duration2, duration>` holds true, the second constructor doesn't participate in overload resolution. For more information, see <type_traits>.

The third constructor initializes its `duration` value by using `Tp.time_since_epoch()`.

time_since_epoch

Returns the amount of time between this `time_point` and the clock's epoch. (A clock's epoch is the time from which the clock starts measuring time, such as midnight 1/1/1970). It's returned as a duration value.

```
C++
```

```
constexpr duration time_since_epoch() const;
```

See also

<chrono>

Header Files Reference

`time_zone` class

Article • 06/15/2022

A `time_zone` represents the time zone for a specific geographic area. It has the full history of time zone rules for each area so time conversions will be accurate if you convert a time for a date back when the rules are different for the time zone than they are today.

Syntax

```
C++
```

```
class time_zone;  // Since C++20
```

Remarks

The `<chrono>` library creates `time_zone` objects as part of its time zone database initialization. It provides **`const`** access to the created objects.

You can't construct or copy a `time_zone` object, and using the default move constructor or default move assignment operator results in undefined behavior.

This is how you get a `time_zone` instance:

```
C++
```

```
const auto& timeZoneDatabase = get_tzdb(); // initialize the time zone
database
const auto& currentZone = timeZoneDatabase.current_zone();
```

Microsoft C++ supports the `time_zone` class starting in Visual Studio 2019 version 16.10. The `time_zone` class is a C++20 feature. The /std:c++latest compiler option is required.

Members

Public member functions and function templates

Name	Description

Name	Description
get_info	Gets a `sys_info` or `local_info` for this `time_zone`.
name	Gets the name of this `time_zone`.
to_local	Converts a `sys_time` to a `local_time` in this `time_zone`.
to_sys	Converts a `local_time` to a `sys_time` in this `time_zone`.

Requirements

Header: `<chrono>`

Time-zone data is only available for Windows 10 version 1903/19H1 and later, and Windows Server 2022 and later.

Namespace: `std::chrono`

`get_info`

The function template `get_info` has two overloads that get a `sys_info` or `local_info` for this `time_zone`.

C++

```
template<class Duration>
sys_info get_info(const sys_time<Duration>& st) const;  // Since C++20

template<class Duration>
local_info get_info(const local_time<Duration>& lt) const;  // Since C++20
```

Parameters

`Duration`
The `duration` class for the `sys_time` or `local_time` parameter.

`st`
The `sys_time` time point used to get a `sys_info` result.

`lt`
The `local_time` time point used to get a `local_info` result.

Return value

In the `get_info` function template that takes a `sys_time` time point `st`, it returns a `sys_info` object `i` such that `st` is in the range `[i.begin, i.end)`.

In the `get_info` function template that takes a `local_time` time point `lt`, it returns a `local_info` object.

Remarks

Microsoft C++ supports `time_zone::get_info` starting in Visual Studio 2019 version 16.10. The function is a C++20 feature that requires the /std:c++latest compiler option.

`name`

Gets the name of this `time_zone`.

C++

```
string_view name() const noexcept;  // Since C++20
```

Return value

Returns the name of the time zone as a `string_view`.

Remarks

Microsoft C++ supports `time_zone::name` starting in Visual Studio 2019 version 16.10.

`to_local`

The function template `to_local` converts a `sys_time` to a `local_time` in this `time_zone`.

C++

```
template<class Duration>
local_time<common_type_t<Duration, seconds>>
to_local(const sys_time<Duration>& st) const;  // Since C++20
```

Parameters

`Duration`

The `duration` class for the `sys_time` or `local_time` parameter.

`st`

The `sys_time` time point used to get a `sys_info` result.

Return value

`to_local` returns the `local_time` associated with *`st`* in this `time_zone`.

Remarks

Microsoft C++ supports `time_zone::to_local` starting in Visual Studio 2019 version 16.10. The function is a C++20 feature that requires the /std:c++latest compiler option.

Example: convert `sys_time` to `local_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    const auto& timeZoneDatabase = get_tzdb();
    const auto& currentZone = timeZoneDatabase.current_zone();
    local_time<system_clock::duration> lt = currentZone-
>to_local(system_clock::now());

    std::cout << "local_time: " << lt << "\n";

    return 0;
}
```

Output

```
local_time: 2021-09-08 15:15:53.1830646
```

to_sys

The function template `to_sys` has two overloads that convert a `local_time` to a `sys_time` in this `time_zone`.

```
C++
```

```
template<class Duration>
sys_time<common_type_t<Duration, seconds>>
to_sys(const local_time<Duration>& lt) const;  // Since C++20

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
to_sys(const local_time<Duration>& lt, choose z) const;  // Since C++20
```

Parameters

Duration
The `duration` class for the `local_time` parameter.

lt
The `local_time` time point to convert.

z
A value of `choose::earliest` or `choose::latest`. It's used to resolve otherwise ambiguous results.

Return value

`to_sys` returns a `sys_time` that's at least as fine as `seconds`. It's finer if the argument *lt* has finer precision. The returned `sys_time` is the UTC equivalent of *lt* according to the rules of this time_zone.

The one-parameter overload throws an `ambiguous_local_time` exception if the conversion from *lt* to a `sys_time` is ambiguous, and a `nonexistent_local_time` exception if the local time point represents a non-existent local time point. Ambiguity can happen during a daylight saving time to standard time transition. Two instances of the same local time point may occur in one day. A non-existent local time point represents a time point in the transition from standard time to daylight saving time.

The two-parameter overload doesn't throw an exception in these cases. If the conversion from *lt* to a `sys_time` is ambiguous, `to_sys` returns the earlier `sys_time` if

`z` is `choose::earliest`, and returns the later `sys_time` if `z` is `choose::latest`. If the `lt` represents a non-existent time between two UTC time points, then the two UTC time points are the same, so `to_sys` returns that UTC time point.

Remarks

Microsoft C++ supports `time_zone::to_sys` starting in Visual Studio 2019 version 16.10. The function is a C++20 feature that requires the /std:c++latest compiler option.

Example: convert `local_time` to `sys_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    const auto& timeZoneDatabase = get_tzdb();
    const auto& currentZone = timeZoneDatabase.current_zone();

    auto st = currentZone->to_sys(local_days{2021y/September/15d}+16h+45min,
choose::earliest);

    std::cout << "sys_time: " << st << "\n";

    return 0;
}
```

Output

```
sys_time: 2021-09-15 23:45:00.0000000
```

See also

<chrono>
Header files reference

`time_zone_link` class

Article • 06/15/2022

Specifies an alternative name for a `time_zone`.

Syntax

```
C++

class time_zone_link;  // Since C++20
```

Remarks

`time_zone_link` instances are created when the time zone database is initialized. They can't be created directly. Get an instance via `get_tzdb().links`, as shown in the example at the end of this topic.

Although this type has a default move constructor and move assignment operator, it isn't copyable because only `const` access is provided to this type. Using those functions results in undefined behavior, which is why the move constructor and assignment operator aren't listed here.

Differences with IANA database

In Microsoft's implementation, time zone data can differ in some cases from the Internet Assigned Numbers Authority (IANA) time zone database. For example, `"America/Nuuk"` is returned instead of `"America/Godthab"` (`"America/Nuuk"` was renamed `"America/Godthab"` in April 2020). We don't supply the `"Eastern War Time EWT"` time zone at this time. For more information, see Issue #1786 on the Microsoft STL GitHub repo.

Members

Public member functions and function templates

Name	Description
name	Gets the alternate name for the time zone.

Name	Description
target	The name of the `time_zone` that this `time_zone_link` is an alternative name for.

Non-members

Name	Description
operator==	Determine whether two `time_zone_link` objects have the same `name`. The `target` name isn't part of the equality check.
operator<=>	Compare this `time_zone_link` against another `time_zone_link`. The `>, >=, <=, <, !=` operators are synthesized by the compiler. Returns `Left.name() <=> Right.name()`.

Requirements

Header: `<chrono>`

Time-zone data is only available for Windows 10 version 1903/19H1 and later, and Windows Server 2022 and later.

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`name`

Gets the alternate name for the time zone.

```
C++

string_view name() const noexcept;
```

`target`

Gets the name of the `time_zone` that this `time_zone_link` is an alternative name for.

```
C++

string_view target() const noexcept;  // Since C++20
```

Example: show `time_zone_link` name

The following example displays the names of each `time_zone_link` in the IANA time zone database.

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    const auto& my_tzdb = get_tzdb(); // get the IANA time zone database
    auto& links = my_tzdb.links; // get the list of time_zone_links from the
database
    std::for_each(links.begin(), links.end(), // display each time_zone_link
name and target
        [](auto& l)
        {
            std::cout << "Name: " << l.name() << "Target" << l.target() <<
'\n';
        });

    return 0;
}
```

Output

```
Name: ACT Target:Australia/Darwin
Name: AET Target:Australia/Sydney
Name: AGT Target:America/Buenos_Aires
Name: ART Target:Africa/Cairo
Name: AST Target:America/Anchorage
Name: Africa/Asmara Target:Africa/Asmera
Name: Africa/Timbuktu Target:Africa/Bamako
Name: America/Argentina/Buenos_Aires Target:America/Buenos_Aires
Name: America/Argentina/Catamarca Target:America/Catamarca
...
Name: Zulu Target:Etc/UTC
```

See also

<chrono>

time_zone

Header files reference

`treat_as_floating_point` structure

Article • 06/15/2022

Checks whether the specified type can be treated as a floating-point type.

Syntax

```
C++
```

```
template <class T>
struct treat_as_floating_point : is_floating_point<T>;
```

Helper variable template

```
C++
```

```
template<class T>
inline constexpr bool treat_as_floating_point_v =
treat_as_floating_point<T>::value;
```

Parameters

`T`

The type to test.

Remarks

The purpose of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If `treat_as_floating_point_v<T>` is false, `T` will be treated as if it behaves as an integral type for these conversions.

The duration template uses the `treat_as_floating_point trait` to help determine if a `duration` object can be converted to another `duration` with a different tick `period`.

If `treat_as_floating_point_v<T>` is true, implicit conversions are allowed between durations. Otherwise, the implicit convertibility depends on the tick periods of the durations.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

See also

<chrono>
Header Files Reference

`tzdb` struct

Article • 06/15/2022

Represents a copy of the time zone database.

Syntax

```
C++
```

```
struct tzdb; // C++ 20
```

Remarks

Represents a copy of the IANA (Internet Assigned Numbers Authority) time zone database. The database contains information about each time zone and its history of local time adjustments. For more information about the database, see Time Zone Database.

Members

Member	Description
`leap_seconds`	A sorted `vector<leap_second>`. A leap_second provides a history of when leap seconds were added in the past.
`links`	A sorted `vector<time_zone_link>`. A time_zone_link provides the alternate name for a given time zone.
`version`	A `std::string` containing the database version.
`zones`	A sorted `vector<time_zone>`. A time_zone has the full history of time zone rules for a particular area.

For an example of how to use these members, see Example: leap_seconds, links, zones near the end of this topic.

Member functions

Name	Description

Name	Description
current_zone	Gets the local time zone.
locate_zone	Gets the specified time zone.

`current_zone`

Gets the computer's local time zone.

C++

```
time_zone* current_zone() const; // C++20
```

Return value

A pointer to the time_zone that represents the local time zone of the computer.

Example: `current_zone`

This example demonstrates getting the current time zone.

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    auto& db = get_tzdb();
    std::cout << "Current zone: " << db.current_zone()->name();

    return 0;
}
```

Output

```
Current zone: America/Los_Angeles
```

`locate_zone`

Gets the specified time zone.

C++

```
const time_zone* locate_zone(string_view name) const;
```

Example: `locate_zone`

This example demonstrates getting a time zone by name.

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    auto& db = get_tzdb();
    std::cout << "Locate zone : " << db.locate_zone("America/New_York")-
>name() << "\n";

    return 0;
}
```

Output

```
Locate zone : America/New_York
```

Example: `leap_seconds, links, zones`

This example demonstrates using various `tzdb` data members.

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    std::cout << "Leap seconds:\n";
```

```
    std::for_each(db.leap_seconds.begin(), db.leap_seconds.end(),
        [](const leap_second& ls)
        {
            std::cout << ls.date() << "\n";
        });

    std::cout << "Links:\n";
    std::for_each(db.links.begin(), db.links.end(),
        [](const time_zone_link& l)
        {
            std::cout << "Name: " << l.name() << "\t\tTarget:" << l.target()
<< '\n';
        });

    std::cout << "\nTime Zone descriptions:\n";
    std::for_each(db.zones.begin(), db.zones.end(),
        [](const time_zone& z)
        {
            std::cout << "Zone: " << z.name() << "\n";
        });

    return 0;
}
```

Output

```
Leap seconds:
1972-07-01 00:00:00
1973-01-01 00:00:00
1974-01-01 00:00:00
...
Links:
Name: ACT               Target:Australia/Darwin
Name: AET               Target:Australia/Sydney
Name: AGT               Target:America/Buenos_Aires
Name: ART               Target:Africa/Cairo
...
Time Zone descriptions:
Zone: Africa/Abidjan
Zone: Africa/Accra
Zone: Africa/Addis_Ababa
...
```

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

See also

<chrono>
tzdb_list
time_zone_link
Header Files Reference

`tzdb_list` class

Article • 06/15/2022

A list of time zone databases.

Syntax

```
C++
```

```
class tzdb_list; // C++ 20
```

Remarks

You can't construct a `tzdb_list`. To get one, call get_tzdb_list().

You can't use copy assignment with a `tzdb_list` as `operator=` is deleted.

`tzdb_list` is a singleton.

Member functions

Function	Description
begin, cbegin	Get an iterator to the start of the list of `tzdb` objects.
end, cend	Get an iterator past the end of the list of `tzdb` objects.
erase_after	Erase the `tzdb` that immediately follows the element pointed to by the passed in iterator.
front	Get the first element in the list of `tzdb` objects.

Member types

Member type	Description
`class const_iterator`	A constant iterator of type `tzdb`.

`begin`, `cbegin`

Gets an iterator that points to the start of the list.

```
C++
```

```
const_iterator begin() const noexcept; // C++20
const_iterator cbegin() const noexcept; // C++20
```

Return value

An iterator pointing to the first `tzdb` in the list.

`end`, `cend`

Get an iterator past the end of the list.

```
C++
```

```
const_iterator end() const noexcept; // C++20
const_iterator cend() const noexcept; // C++20)
```

Return value

An iterator pointing past the end of the last `tzdb` in the list.

Remarks

Because the returned iterator points past the end of the list, it shouldn't be dereferenced. Instead, use it to compare when you've reached the end of the list.

`front`

Gets the first tzdb in the list.

```
C++
```

```
const tzdb& front() const noexcept; // C++20
```

Return value

The first time zone database in the list.

Remarks

You can simultaneously call this function and reload_tzdb without risk of a race condition.

`erase_after`

Erase the `tzdb` that follows the element pointed to by the passed-in iterator.

```
C++

const_iterator erase_after(const_iterator p);
```

Parameters

p
An iterator that points to the element before the one to erase.

Return value

An iterator following the erased `tzdb`. If there isn't an element following the erased `tzdb`, then `end()`

Remarks

It isn't possible to erase the first element in the list.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

See also

<chrono>

tzdb struct

Header Files Reference

`utc_clock` class

Article • 06/15/2022

A clock for Coordinated Universal Time (UTC) which is sometimes popularly referred to as Greenwich Mean Time (GMT) or Zulu time. Local time is adjusted from UTC time depending on your time zone.

Syntax

```
C++
```

```
class utc_clock; // C++20
```

Remarks

This clock measures time since 00:00:00 UTC on Thursday, January 1, 1970. This clock accounts for leap seconds and is the basis for civil time around the world.

The rate of UTC time is based on International Atomic Time (TAI) which uses atomic clocks to track time. However, a leap second is occasionally added to UTC time to keep the difference between UTC time and UT1 (solar time) within +- 0.9 seconds.

There's a discrepancy between the time kept by atomic clocks, and the time kept by tracking the rotation of the earth. That's because the earth's rotation speed is irregular, and in general is slowing down over time by about one second every 1.5 years (although sometimes the earth's rotation speed increases). A leap second adjustment keeps UTC tracking time within a second of the average length of a day.

As of this writing, 27 leap seconds have been added since the practice of inserting leap seconds began in 1972. The International Earth Rotation and Reference Systems Service (IERS) determines when a leap second will be added. Adding a leap second is referred to as a "leap second insertion". When a leap second is inserted, the time, as it nears midnight, proceeds from 23 hours 59 minutes 59 seconds to 23 hours 59 minutes 60 seconds (the inserted leap second), and then to 0 hours 0 minutes 0 seconds (midnight). Historically, leap seconds have been added on June 30 or December 31.

UTC time, by definition, starts out 10 seconds behind TAI (atomic time). 10 seconds were added in 1972 to TAI time to accommodate for the leap seconds that had accumulated by that point. Given the insertion of another 27 leap seconds since then, UTC time is currently 37 seconds behind TAI (atomic clock) time.

Members

Name	Description
from_sys	Static. Converts a `sys_time` to a `utc_time`.
now	Static. Returns the current time.
to_sys	Static. Converts a `utc_time` to a `sys_time`.

Non-members

Name	Description
from_stream	Parse a `utc_time` from the given stream using the specified format.
get_leap_second_info	Gets a leap_second_info that indicates whether the supplied time specifies a time when a leap second was inserted. It also includes the sum of all the leap seconds between January 1, 1970 and the specified duration.
operator<<	Output a `utc_time` to the given stream.

Convenience type aliases

Name	Description
`utc_clock::duration`	In Microsoft's implementation, it's a synonym for `duration<system_clock::rep, system_clock::period>`. It represents a duration of time measured in units of 100 nanoseconds.
`utc_clock::period`	Microsoft's implementation defines this as a synonym for `system_clock::period`. It represents the time in seconds (100 nanoseconds) between each tick in the duration.
`utc_clock::rep`	A synonym for the type (`long long`) used to represent the integral units in this clock's `utc_clock::duration`.
`utc_clock::time_point`	A synonym for `time_point<utc_clock>`. Used to represent a time_point for this clock.

Related

Name	Description
`utc_seconds`	A synonym for `using utc_seconds = utc_time<seconds>;`

Name	Description
`utc_time`	A synonym for `template<class Duration> using utc_time = time_point<utc_clock, Duration>;` Represents a time_point for a `utc_clock`. You specify the duration. Defined in `std::chrono`

Public Constants

Name	Description
[`utc_clock::is_steady constant`] (#is_steady_constant)	Indicates whether the clock type is steady. Its value is `false`.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

`from_sys`

Static method that converts a `sys_time` to a `utc_time`.

```
C++
```

```
template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
from_sys(const sys_time<Duration>& t);
```

Parameters

t

The `sys_time` to convert.

Return value

A `utc_time` set to the time that most closely approximates the `sys_time` *t*. If a direct mapping doesn't exist, the time_point during a leap second insertion and the last

representable value of `sys_time` before the insertion of the leap second, is returned.

`to_sys`

Static method that converts a `utc_time` to a `sys_time`.

```
C++
```

```
template<class Duration>
static sys_time<common_type_t<Duration, seconds>>
to_sys(const utc_time<Duration>& t);
```

Parameters

t
The `utc_time` to convert.

Return Value

The `sys_time` set to the equivalent point in time as `t`. If a direct mapping doesn't exist, it's the last representable value before the insertion of a leap second in the case that `t` represents a point in time when a leap second was inserted. Adding a leap second is referred to as a "leap second insertion". When a leap second is inserted, the time as it nears midnight proceeds from 23 hours 59 minutes 59 seconds to 23 hours 59 minutes 60 seconds (the inserted leap second), and then to 0 hours 0 minutes 0 seconds (midnight). In the past, leap seconds have been added either on June 30 or December 31.

`is_steady`

Static value that specifies whether the clock type is *steady*. In Microsoft's implementation, `is_steady_constant` is **`false`**. Because the `utc_clock` isn't steady, you can't reliably use this clock to take the time before an event, the time after an event, and subtract them to get the duration of the event because the clock may be adjusted during that time.

```
C++
```

```
static const bool is_steady = false;
```

now

Static method that returns the current UTC time. The returned time includes leap seconds.

```
C++
```

```
static time_point now() noexcept;
```

Return Value

A time_point object that represents the current time. The returned time point is effectively `from_sys(system_clock::now())`.

See also

<chrono>
file_clock class
gps_clock class
high_resolution_clock
local_t struct
steady_clock struct
system_clock struct
tai_clock class
Header Files Reference

`weekday` class

Article • 08/03/2021

Represents a day of the week in the Gregorian calendar. For example, Tuesday.

Syntax

```
C++
```

```
class weekday; // C++20
```

Remarks

A `weekday` can hold a value of [0, 255], but typically holds a value [0, 6] to represent a day of the week. See Weekday constants, below, for constants that you can use with the `weekday` class.

Members

Name	Description
Constructors	Construct a `weekday`.
c_encoding	Get the `weekday` value.
iso_encoding	Get the ISO 8601 `weekday` value.
ok	Check if the `weekday` value is valid.
operator++	Increment the `weekday`.
operator+=	Add the specified number of weekdays to this `weekday`.
operator--	Decrement the `weekday`.
operator-=	Subtract the specified number of weekdays from this `weekday`.
operator[]	Create a weekday_indexed or weekday_last from this `weekday`.

Non-members

Name	Description
from_stream	Parse a `weekday` from the given stream using the specified format.
operator+	Add specified number of weekdays to this `weekday`.
operator-	Subtract the specified number of weekdays from this `weekday`, or find the difference between two `weekday` objects.
operator==	Determine whether two `weekday` objects are equal.
operator<<	Output a `weekday` to a stream.

Requirements

Header: `<chrono>` Since C++ 20

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Constructs a `weekday`.

```
C++

1) weekday() = default; // C++20
2) explicit constexpr weekday(unsigned wd) noexcept; // C++20
3) constexpr weekday(const sys_days& dp) noexcept; // C++20
4) explicit constexpr weekday(const local_days& dp) noexcept; // C++20
```

Parameters

dp

Construct a `weekday` using the day of the week *dp*.

wd

Construct a `weekday` with value *wd*.

Remarks

1) The default constructor doesn't initialize the weekday value.
2) Construct a `weekday` with the specified value.
If `wd` is 7, the `weekday` is constructed with a value of 0.
Don't initialize with values over 255 or the resulting `weekday` will have an unspecified value.
3) Computes what day of the week corresponds to the `std::chrono::sys_days` value `dp`, and constructs a `weekday` using that day.
4) Computes the day of the week that corresponds to the `std::chrono::local_days` value `dp`, and constructs a `weekday` using that day. It behaves as if you created the `weekday` using `weekday(std::chrono::sys_days(dp.time_since_epoch()))`.

Example: Create a `weekday`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    weekday wd{ Wednesday };
    weekday wd2{ 3 };
    std::cout << wd << '\n' << wd2;

    return 0;
}
```

Output

```
Wednesday
Wednesday
```

`c_encoding`

C++

```
constexpr unsigned c_encoding() const noexcept;
```

Return value

The weekday value.

`iso_encoding`

The weekday value, but with Sunday (0) is interpreted as 7 per ISO 8601.

```
C++
```

```
constexpr unsigned iso_encoding() const noexcept;
```

Return value

The weekday value.

`ok`

Check if the value stored in this `weekday` is in the valid range.

```
C++
```

```
constexpr bool is_leap() const noexcept;
```

Return value

`true` if the weekday value is in the range [0, 6]. Otherwise, `false`.

`operator++`

Add 1 to the weekday value.

```
C++
```

```
1) constexpr std::chrono::weekday& operator++() noexcept;
2) constexpr std::chrono::weekday operator++(int) noexcept;
```

Return value

1) A reference to `*this` weekday *after* it has been incremented (a postfix increment).
2) A copy of the `weekday`, *before* it has been incremented (a prefix increment).

Example: `operator++`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
    std::chrono::weekday wd{Thursday};

    std::cout << wd << " " << ++wd << "\n"; // constexpr weekday& operator++
() noexcept
    std::cout << wd << " " << wd++ << "\n"; // constexpr weekday operator++
(int) noexcept
    std::cout << wd << "\n";

    return 0;
}
```

Output

```
Thu Fri
Fri Fri
Sat
```

Remarks

The incremented value is modulo 7. It will always be in the range [0, 6].

`operator--`

Subtract 1 from the `weekday` value.

C++

```
1) constexpr std::chrono::weekday& operator--() noexcept;
2) constexpr std::chrono::weekday operator--(int) noexcept;
```

Return value

1) A reference to `*this` `weekday` *after* it has been decremented (a postfix decrement).
2) A copy of the `weekday` *before* it has been decremented (a prefix decrement).

Example: `operator--`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
    weekday y = weekday{Thursday};

    cout << y << " " << --y << "\n"; // constexpr weekday& operator--()
noexcept
    cout << y << " " << y-- << "\n"; // constexpr weekday operator--(int)
noexcept
    cout << y << "\n";

    return 0;
}
```

Output

```
Thu Wed
Wed Wed
Tue
```

Remarks

If the decremented result is less than 0, it's set to 6.

`operator+=`

Add days to a `weekday`.

C++

```
constexpr weekday& operator+=(const days& d) noexcept;
```

Parameters

`*d*`

The number of days to add.

Return value

The value of `*this + d`. The result will be modulo 7, in the range [0, 6].

`operator-=`

Subtract days from the `weekday`.

```
C++
```

```
constexpr weekday& operator-=(const days& d) noexcept;
```

Parameters

`*d*`

The number of days to subtract.

Return value

The value of `*this - d`. The result will be modulo 7, in the range [0, 6].

`operator[]`

Create a weekday_indexed or weekday_last from this `weekday`.

```
C++
```

```
1) std::chrono::weekday_indexed(*this, index) // C++ 20
2) std::chrono::weekday_last(*this) // C++ 20
```

Return value

1) `std::chrono::weekday_indexed(*this, index)`

2) `std::chrono::weekday_last(*this)`

Example: `operator[]`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std;
using namespace std::chrono;

int main()
{
    constexpr auto firstMondayInJanuary =
        year_month_day{ Monday[2] / January / 2021y };

    std::cout << firstMondayInJanuary << "\n";

    constexpr auto lastMondayInJanuary =
        year_month_day{ Tuesday[last] / January / 2021y };
    std::cout << lastMondayInJanuary << "\n";
}
```

Output

```
2021-01-11
2021-01-26
```

Weekday constants

(C++20) The `<chrono>` header defines the following constants that you can use with `weekday` for greater convenience, type-safety, and maintainability of your code. These constants are in scope when `std::chrono` is in scope.

C++

```
// Calendrical constants
inline constexpr std::chrono::weekday Sunday{0};
inline constexpr std::chrono::weekday Monday{1};
inline constexpr std::chrono::weekday Tuesday{2};
inline constexpr std::chrono::weekday Wednesday{3};
inline constexpr std::chrono::weekday Thursday{4};
inline constexpr std::chrono::weekday Friday{5};
inline constexpr std::chrono::weekday Saturday{6};
```

See also

weekday_last class
weekday_indexed class
<chrono>
Header Files Reference

`weekday_indexed` class

Article • 09/23/2021

Combines a weekday, representing a day of the week in the Gregorian calendar, with an index in the range [1, 5] that represents the weekday of the month (1st, 2nd, 3rd, and so on).

Syntax

```
C++
```

```
class weekday_indexed; // C++20
```

Remarks

`weekday_indexed` is a trivially copyable and standard-layout class type.

Members

Name	Description
Constructors	Construct a `weekday_indexed` with the specified weekday and value.
ok	Check if the weekday value is valid.
weekday	Get the weekday value.

Non-members

Name	Description
operator==	Determine whether two `weekday_indexed` instances are equal.
operator<<	Output a `weekday_indexed` to the specified stream.

Requirements

Header: `<chrono>` Since C++ 20

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `weekday_indexed` initialized with a `weekday` value and an index.

C++

```
constexpr weekday_indexed(const chrono::weekday& wd, unsigned index)
noexcept; // C++20
```

Parameters

wd

The weekday value for the created `weekday_indexed` class.

index

The weekday of the month that the weekday lands on. It is in the range [1, 5]. For example, 2 indicates that the weekday lands on the second week of the month.

Example: Create a `weekday_indexed`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
   constexpr auto wdi = weekday_indexed{Monday, 1};
   constexpr auto wdi2 = Monday[1];

    std::cout << wdi << ", " << wdi2;
    return 0;
}
```

Output

```
Mon[1], Mon[1]
```

ok

Check if the value stored in this `weekday_indexed` is in the valid range.

```
C++
```

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the week weekday value is in the valid range. Otherwise, `false`.

weekday

Get the weekday value.

```
C++
```

```
constexpr chrono::weekday weekday() const noexcept;
```

Return value

The weekday value.

Example

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

int main()
{
    constexpr auto wdi = weekday_indexed{ Monday, 1 };
    std::cout << wdi << "\n";

    return 0;
}
```

```
Output
```

```
Mon[1]
```

See also

weekday class
weekday_last class
<chrono>
Header Files Reference

`weekday_last` class

Article • 01/04/2024

Represents the last weekday of a month.

Syntax

```
C++
```

```
class weekday_last; // C++20
```

Remarks

`weekday_last` is a trivially copyable and standard-layout class type.

Members

Constructor

Expand table

Name	Description
weekday_last	Constructs a `weekday_last` with the specified weekday value.

Functions

Expand table

Name	Description
ok	Check if the weekday value is valid.
weekday	Get the weekday value.

Operators

Expand table

Name	Description
operator==	Determine whether two `weekday_last` instances are equal.
operator<<	Output a `weekday_last` to the specified stream.

Requirements

Header: `<chrono>` Since C++20

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `weekday_last` that is initialized with a `weekday` value.

```
C++
```

```
constexpr explicit weekday_last(const chrono::weekday& wd) noexcept; //
C++20
```

Parameters

wd
The weekday value for the created `weekday_last` class.

Remarks

It's easier to construct a `weekday_last` using the `weekday` `operator[]`. See the example below.

ok

Check if the value stored in this `weekday_last` is in the valid range.

```
C++
```

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the week weekday value is in the valid range. Otherwise, `false`.

`weekday`

Get the weekday value.

C++

```
constexpr chrono::weekday weekday() const noexcept;
```

Return value

The weekday value.

Example

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    constexpr auto wdl{ Monday[last] }; // wdl is the last Monday of an
unspecified month
    std::cout << wdl.weekday() << "\n";

    return 0;
}
```

Output

```
Mon
```

See also

<chrono>
weekday class

`year` class

Article • 09/23/2021

Represents a year in the Gregorian calendar.

Syntax

```
C++
```

```
class year; // C++20
```

Remarks

A `year` can hold a year value between -32767 to 32767.

Members

Name	Description
Constructors	Construct a `year`
is_leap	Determine if the year is a leap year.
max	Returns the largest possible year value.
min	Returns the smallest possible year value.
ok	Verify that the year value is in the valid range [-32767, 32767].
operator+	Unary plus.
operator++	Increment the year.
operator+=	Add the specified number of years to this `year`.
operator-	Unary minus.
operator--	Decrement the year.
operator-=	Subtract the specified number of years from this `year`.
operator int	Get the `year` value.

Non-members

Name	Description
from_stream	Parse a `year` from a stream using the specified format
operator+	Add years.
operator-	Subtract years.
operator==	Determine whether two years are equal.
operator<=>	Compare this `year` against another `year`. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `year` to the given stream.
operator""y	Create a `year` literal.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `year`.

```
C++

1) year() = default;
2) explicit constexpr year(unsigned y) noexcept;
```

Parameters

y

Construct a `year` with value *y*.

Remarks

1) The default constructor doesn't initialize the `year` value.

2) Construct a `year` with the specified value.

Example: Create a `year`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year y{2020};
    year y2 = 2021y;

    std::cout << y << ", " << y2;

    return 0;
}
```

Output

```
2020, 2021
```

`is_leap`

Check if the value stored in this `year` is in the valid range.

C++

```
constexpr bool is_leap() const noexcept;
```

Return value

`true` if the year value is a leap year. Otherwise, `false`. A leap year is a year divisible by 4 but not 100--or is divisible by 400.

`max`

Returns the largest possible year.

```
C++
```

```cpp
static constexpr year max() noexcept;
```

Return value

`year{32767}`

`min`

Returns the smallest possible year.

```
C++
```

```cpp
static constexpr year min() noexcept;
```

Return value

`year{-32767}`

`ok`

Check if the value stored in this `year` is in the valid range.

```
C++
```

```cpp
constexpr bool ok() const noexcept;
```

Return value

`true` if the year value is in the range [-32676, 32767]. Otherwise, `false`.

`operator+`

Apply unary plus.

```
C++
```

```cpp
constexpr year operator+() const noexcept;
```

Return value

Returns `*this`

Example: unary `operator+`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
   year y{-1};
   std::cout << +y;
   return 0;
}
```

Output

```
-0001
```

`operator++`

Add 1 to the year value.

C++

```
1) constexpr year& operator++() noexcept;
2) constexpr year operator++(int) noexcept;
```

Return value

1) Returns reference to this year *after* it has been incremented (a postfix increment).
2) Returns a copy of the `year`, *before* it has been incremented (a prefix increment).

Example: `operator++`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year y{2021};

    std::cout << y << " " << ++y << "\n"; // constexpr year& operator++()
noexcept
    std::cout << y << " " << y++ << "\n"; // constexpr year operator++(int)
noexcept
    std::cout << y << "\n";
    return 0;
}
```

Output

```
2021 2022
2022 2022
2023
```

Remarks

If the incremented result exceeds 32767, it overflows to -32768

`operator-`

Unary minus. Negate the `year`.

C++

```
constexpr year operator-() const noexcept; // C++ 20
```

Return value

Returns a negated copy of the `year`.

Example: unary `operator-`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
   year y{1977};
   std::cout << -y << '\n';

   return 0;
}
```

Output

```
-1977
```

`operator--`

Subtract 1 from the year value.

C++

```
1) constexpr year& operator--() noexcept;
2) constexpr year operator--(int) noexcept;
```

Return value

1) A reference to this `year` *after* it has been decremented (a postfix decrement).

2) A copy of the `year` *before* it has been decremented (a prefix decrement).

Example: `operator--`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
   year y{2021};
```

```
    std::cout << y << " " << --y << "\n"; // constexpr year& operator++()
noexcept
    std::cout << y << " " << y-- << "\n"; // constexpr year operator++(int)
noexcept
    std::cout << y << "\n";

    return 0;
}
```

Output

```
2021 2020
2020 2020
2019
```

Remarks

If the decremented result is less than -32768, it's set to 32767.

`operator+=`

Add days to this `year`.

C++

```
constexpr year& operator+=(const years& y) noexcept;
```

Parameters

`y`
The number of years to add.

Return value

`*this` If the incremented result exceeds 32767, it overflows to -32768.

`operator-=`

Subtract days from this `year`.

C++

```
constexpr year& operator-=(const years& y) noexcept;
```

Parameters

`y`
The number of years to subtract.

Return value

`*this`. If the decremented result is less than -32768, it's set to 32767.

`operator int`

Get the `year` value.

C++

```
explicit constexpr operator int() const noexcept;
```

Return value

The value of the `year`

Example: `operator int()`

C++

```
// compile using: /std:c++latest

 #include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year y{2020};
    int yearValue = static_cast<int>(y);
    std::cout << yearValue;

    return 0;
}
```

Output

```
2020
```

See also

<chrono>
year_month
year_month_day
year_month_day_last
year_month_weekday
year_month_weekday_last
Header Files Reference

`year_month` class

Article • 06/15/2022

Represents a month and year. The day isn't specified.

Syntax

```
C++
```

```
class year_month; // C++20
```

Members

Name	Description
Constructors	Construct a `year_month`
year	Returns the year.
month	Returns the month.
ok	Verify that the year and month values are in the valid range.
operator+=	Add the specified number of months or years.
operator-=	Subtract the specified number of months or years.

Non-members

Name	Description
from_stream	Parse a `year_month` from a stream using the specified format
operator+	Add months and/or years.
operator-	Subtract months and/or years.
operator==	Determine whether two `year_month` values are equal.
operator<=>	Compare two `year_month` values. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `year_month` to a stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `year_month`.

```
C++

1) year_month() = default;
2) constexpr year_month(const year& y, const month& m) noexcept;
```

Parameters

y

The year value.

m

The month value.

Remarks

1) The default constructor doesn't initialize the `year` or `month` value.

2) Construct a `year_month` with the specified values.

For information about C++ 20 syntax to specify dates, see operator/

Example: Create a `year_month`

```
C++

// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
```

```
{
    year_month ym{2021y / June};

    std::cout << ym;
    return 0;
}
```

Output

```
2021/Jun
```

`month`

Get the month.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`year`

Get the year.

C++

```
constexpr year year() const noexcept;
```

Return value

The year.

`ok`

Check if the year and month value stored in this `year_month` are both in the valid range.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the `year_month` year and month values are in the valid range. Otherwise, `false`.

`operator+=`

Add months or years to this `year_month`.

C++

```
1) constexpr year_month& operator+=(const months& dm) noexcept;
2) constexpr year_month& operator+=(const years& dy) noexcept;
```

Parameters

dm

The number of months to add.

dy

The number of years to add.

Return value

`*this`, which reflects the result of the addition.

Example: `operator +=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month ym{2021y / June};

    std::cout << ym << '\n';
```

```
    ym += months{2};
    ym += years{1};

    std::cout << ym;

    return 0;
}
```

Output

```
2021/Jun
2022/Aug
```

`operator-=`

Subtract months or years from this `year_month`.

C++

```
1) constexpr year_month& operator-=(const months& dm) noexcept;
2) constexpr year_month& operator-=(const years& dy) noexcept;
```

Parameters

dm

The number of months to subtract.

dy

The number of years to subtract.

Return value

`*this`, which reflects the result of the subtraction.

Example: `operator -=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>
```

```
using namespace std::chrono;

int main()
{
    year_month ym{2021y / June};

    std::cout << ym << '\n';

    ym -= months{2};
    ym -= years{1};

    std::cout << ym;

    return 0;
}
```

Output

```
2021/Jun
2020/Apr
```

See also

<chrono>
year
year_month_day
year_month_day_last
year_month_weekday
year_month_weekday_last
operator/
Header Files Reference

`year_month_day` class

Article • 08/05/2024

Represents a month, year, and day.

Syntax

```
C++
```

```
class year_month_day; // C++20
```

Members

Expand table

Name	Description
Constructors	Construct a `year_month_day`
day	Returns the day.
month	Returns the month.
ok	Verify that the year and month values are in the valid range.
operator+=	Add the specified number of months or years.
operator-=	Subtract the specified number of months or years.
operator local_days	Get the count of days from the `system_clock` epoch to this `year_month_day` as local_days.
operator sys_days	Get the count of days from the `system_clock` epoch to this `year_month_day` as sys_days.
year	Returns the year.

Non-members

Expand table

Name	Description
from_stream	Parse a `year_month_day` from a stream using the specified format
operator+	Add months or years.
operator-	Subtract months or years.
operator==	Determine whether two `year_month_day` values are equal.
operator<=>	Compare two `year_month_day` values. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `year_month_day` to a stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `year_month_day`.

C++

```
1) year_month_day() = default;
2) constexpr year_month_day(const year& y, const month& m, day& d) noexcept;
3) constexpr year_month_day(const year_month_day_last& ymdl) noexcept;
4) constexpr year_month_day(const sys_days& dp) noexcept;
5) constexpr explicit year_month_day(const local_days& dp) noexcept;
```

Parameters

d A day value.

dp A sys_days or local_days value.

m A month value.

y
A year value.

ymdl A `year_month_day_last` value.

Remarks

1) The default constructor doesn't initialize the month or day.
2) Constructs a `year_month_day` with the specified year, month, and day.
3) Constructs a `year_month_day` with the specified year, month, and day from *ymdl*
4) Constructs a `year_month_day` with the same date as *dp*.
5) Constructs a `year_month_day` with the same date as *dp* but as though constructed by `year_month_day(sys_days(dp.time_since_epoch()))`.

For information about C++20 syntax used to specify dates, see operator/

Example: Create a `year_month_day`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day ymd{ April / 4 / 1975 };
    std::cout << ymd;
    return 0;
}
```

Output

```
1975-04-04
```

day

Get the day.

C++

```
constexpr day day() const noexcept;
```

Return value

The day value.

`month`

Get the month.

```
C++
```

```
constexpr month month() const noexcept;
```

Return value

The month value.

`operator local_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_day` as local_days

```
C++
```

```
constexpr explicit operator local_days() const noexcept;
```

Return value

If `ok()`, returns a count of days as `local_days{sys_days{*this}.time_since_epoch()}`

`operator sys_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_day` as sys_days.

```
C++
```

```
constexpr operator sys_days() const noexcept;
```

Return value

If `ok()`, returns a `sys_days` holding a count of days from the `sys_days` epoch (1/1/1970) to the date held in this `year_month_day`. The value will be negative if the date in this `year_month_day` is prior to the `sys_days` epoch.

If the year and month in this `year_month_day` are `ok()`, returns `sys_days{year/month/1d} + (day-1d)`. Otherwise the value returned is unspecified.

A `sys_days` in the range [`days{-12687428}`, `days{11248737}`] can be converted to a `year_month_day` and back and have have the same value.

year

Get the year.

```
C++
constexpr year year() const noexcept;
```

Return value

The year.

ok

Check if the year and month value stored in this `year_month_day` are both in the valid range. Ensures that the day is in the range [1d, (y/m/last).day()], accounting for leap years and the different number of days in each month.

```
C++
constexpr bool ok() const noexcept;
```

Return value

`true` if the `year_month_day` year, month, and day values are in the valid range. Otherwise, `false`.

operator+=

Add months or years to this `year_month_day`.

C++

```
1) constexpr year_month_day& operator+=(const months& dm) noexcept;
2) constexpr year_month_day& operator+=(const years& dy) noexcept;
```

Parameters

dm

The number of months to add.

dy

The number of years to add.

Return value

`*this`, which reflects the result of the addition.

Example: `operator+=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day ymd{June / 1d / 2021y};

    std::cout << ymd << '\n';

    ymd += months{2};
    ymd += years{1};

    std::cout << ymd;

    return 0;
}
```

Output

```
2021-06-01
2022-08-01
```

`operator-=`

Subtract months or years from this `year_month_day`.

C++

```
1) constexpr year_month_day& operator-=(const months& dm) noexcept;
2) constexpr year_month_day& operator-=(const years& dy) noexcept;
```

Parameters

dm

The number of months to subtract.

dy

The number of years to subtract.

Return value

`*this`, which reflects the result of the subtraction.

Example: `operator-=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day ymd{June / 1d / 2021y};

    std::cout << ymd << '\n';

    ymd -= months{2};
    ymd -= years{1};

    std::cout << ymd;
```

```
    return 0;
}
```

Output

```
2021-06-01
2020-04-01
```

See also

<chrono>
year
year_month
year_month_day_last
year_month_weekday
year_month_weekday_last
operator/
Header Files Reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`year_month_day_last` class

Article • 06/15/2022

Represents the last day of a specific year and month.

`year_month_day_last` supports month arithmetic and years arithmetic, but not days arithmetic. To do days-oriented arithmetic, convert the `year_month_day_last` to a `sys_days`.

Syntax

```
C++
```

```
class year_month_day_last; // C++20
```

Members

Name	Description
Constructors	Construct a `year_month_day_last`
day	Get the last day of the month/year.
month	Get the month.
month_day_last	Get the month_day_last value stored in this `year_month_day_last`.
ok	Verify that the year and month values are in the valid range.
operator+=	Add months or years to this `year_month_day_last`.
operator-=	Subtract months or years from this `year_month_day_last`.
operator local_days	Get the count of days from the `sys_days` epoch to this `year_month_day_last` as local_days.
operator sys_days	Get the count of days from the `sys_days` epoch to this `year_month_day_last` as sys_days.
year	Get the year.

Non-members

Name	Description
operator+	Add months or years.
operator-	Subtract months or years.
operator==	Determine whether two `year_month_day_last` values are equal.
operator<=>	Compare two `year_month_day_last` values. The `>, >=, <=, <, !=` operators are synthesized by the compiler.
operator<<	Output a `year_month_day_last` to a stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `year_month_day_last` for a specified month and year.

```
C++
```

```
constexpr year_month_day_last(const year& y, const month_day_last& mdl)
noexcept;
```

Parameters

`mdl` The month value from the month_day_last is stored in the constructed `year_month_day_last`.

`y`
The year value is stored in the constructed `year_month_day_last`.

Remarks

For information about C++ 20 syntax used to specify dates, see operator/

Example: Create a `year_month_day_last`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day_last ymdl{ April / last / 1975 };
    std::cout << ymdl;

    return 0;
}
```

Output

```
1975/Apr/last
```

`day`

If `ok()` is `true`, returns a day representing the last day of the year, month represented by this `year_month_day_last`.

C++

```
constexpr day day() const noexcept;
```

Return value

If `ok()` is `true`, returns a day representing the last day of the year and month represented by `*this`. Otherwise, the return value is unspecified.

`local_days`

Get the month, day, and year in this `year_month_day_last` as a count of days.

C++

```
constexpr explicit operator local_days() const noexcept;
```

Return value

`local_days{sys_days{*this}.time_since_epoch()}`

Example

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day_last ymdl{ June / last / 2021 };
    auto ld = local_days(ymdl);
    std::cout << ld.time_since_epoch();

    return 0;
}
```

Output

```
18808d
```

`month`

Get the stored month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`month_day_last`

Gets the `month_day_last` value stored in this `year_month_day_last`.

```
C++
```

```
constexpr month_day_last month_day_last() const noexcept;
```

Return value

The month_day_last value stored in this `year_month_day_last`.

`operator sys_days`

Get the month, day, and year in this `year_month_day_last` as a count of days from the epoch for the system clock.

```
C++
```

```
constexpr explicit operator sys_days() const noexcept;
```

Return value

Returns `sys_days{this->year()/this->month()/this->day()}`

Example

```
C++
```

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day_last ymdl{ June / last / 2021 };
    auto sd = sys_days(ymdl);
    std::cout << sd.time_since_epoch();

    return 0;
}
```

```
Output
```

```
18808d
```

year

Get the year.

```
C++
constexpr year year() const noexcept;
```

Return value

The year.

ok

Check if the `year` and `month_day_last` value stored in this `year_month_day_last` are both in the valid range.

```
C++
constexpr bool ok() const noexcept;
```

Return value

`true` if the `year` and `month_day_last` value stored in this `year_month_day_last` are both in the valid range. Otherwise, `false`.

operator+=

Add months or years to this `year_month_day_last`.

```
C++
1) constexpr year_month_day_last& operator+=(const months& m) noexcept;
2) constexpr year_month_day_last& operator+=(const years& y) noexcept;
```

Parameters

m

The number of months to add.

y

The numbers of years to add.

Return value

`*this`, which reflects the result of the addition.

Example: `operator+=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day_last ymdl{ April / last / 1975 };

    std::cout << ymdl << '\n';

    ymdl += months{1};
    ymdl += years{1};

    std::cout << ymdl;

    return 0;
}
```

Output

```
1975/Apr/last
1976/May/last
```

`operator-=`

Subtract months or years from this `year_month_day_last`.

C++

```
1) constexpr year_month_day_last& operator-=(const months& m) noexcept;
2) constexpr year_month_day_last& operator-=(const years& y) noexcept;
```

Parameters

`m`

The number of months to subtract.

`y`

The number of years to subtract.

Return value

`*this`, which reflects the result of the subtraction.

Example: `operator-=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_day_last ymdl{ April / last / 1975 };

    std::cout << ymdl << '\n';

    ymdl -= months{1};
    ymdl -= years{1};

    std::cout << ymdl;

    return 0;
}
```

Output

```
1975/Apr/last
1974/Mar/last
```

See also

<chrono>
year
year_month
year_month_day
year_month_weekday
year_month_weekday_last
operator/
Header Files Reference

`year_month_weekday` class

Article • 06/15/2022

Represents a specific year, month, and nth weekday of the month.

Syntax

```
C++
```

```
class year_month_weekday; // C++ 20
```

Remarks

`year_month_weekday` supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For days-oriented arithmetic, use the `sys_days` conversion to convert to a sys_days, which supports days-oriented arithmetic.

`year_month_weekday` is a trivially copyable and standard-layout class type.

Members

Name	Description
Constructor	Construct a `year_month_weekday` with the specified month and weekday.
index	Get the index of the weekday.
month	Get the month value.
ok	Check if the `year_month_weekday` is valid.
operator+=	Add the specified number of months or years.
operator-=	Subtract the specified number of months or years.
operator local_days	Get the count of days from the `system_clock` epoch to this `year_month_weekday` as local_days.
operator sys_days	Get the count of days from the `system_clock` epoch to this `year_month_weekday` as sys_days.
weekday	Get the weekday.

Name	Description
weekday_indexed	Get the [`weekday_indexed`] stored in this `year_month_weekday`.
year	Get the year.

Non-members

Name	Description
operator+	Add months or years.
operator-	Subtract months or years.
operator==	Determine whether two `year_month_weekday` values are equal.
operator<<	Output a `year_month_weekday` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `year_month_weekday`.

```
C++
```

```
// 1)
year_month_weekday() = default

// 2)
constexpr year_month_weekday(const year& y, const month& m, const
weekday_indexed& wdi) noexcept;

// 3)
constexpr explicit year_month_weekday(const local_days& dp) noexcept;

// 4)
constexpr year_month_weekday(const sys_days& dp) noexcept;
```

Parameters

m

The month value.

dp

A sys_days or local_days

wdi

The weekday value.

y

The year value.

Remarks: Constructor

1) The default constructor doesn't initialize any of the fields.

2) Constructs a `year_month_weekday` that corresponds to the specified `year`, `month`, and `weekday_indexed`.

3) Constructs a `year_month_weekday` that corresponds to the date represented by `sys_days{dp.time_since_epoch()}`.

4) Constructs a `year_month_weekday` that corresponds to the date represented by *dp*. For any `year_month_weekday` (ymdl) for which `ok()` is `true`, comparison with `operator==` to `year_month_weekday{sys_days{ymdl}}` will be `true`.

For information about C++ 20 syntax used to specify dates, see operator/

Example: Create a `year_month_weekday`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_weekday ymw{1997y / January / Wednesday[1]};
    std::cout << ymw << '\n';
```

```
    return 0;
}
```

Output

```
1997/Jan/Wed[1]
```

`index`

Get the week index of the weekday in this `year_month_weekday`.

C++

```
constexpr unsigned index() const noexcept;
```

Return value

The index of the weekday. For example, if the weekday was the first Wednesday of the week, the index would be 1.

`month`

Get the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`ok`

Check if the value stored in this `year_month_weekday` is valid. The `year`, `month`, and `weekday_index` stored in this `year_month_weekday` must all be `ok` for this function to return `true`. Otherwise, returns `false`.

C++

```
constexpr bool ok() const noexcept;
```

Return value

`true` if the `year_month_weekday` value is valid. Otherwise, `false`.
A `year_month_weekday` is valid if both the `month` is valid and the `weekday_indexed` value is valid.

`operator+=`

Add months or years to this `year_month_weekday`.

C++

```
1) constexpr year_month_weekday& operator+=(const months& m) noexcept;
2) constexpr year_month_weekday& operator+=(const years& y) noexcept;
```

Parameters

m
The number of months to add.

y
The number of years to add.

Return value

`*this` which reflects the result of the addition.

Example: `operator+=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_weekday ymw{1997y / January / Wednesday[1]};
```

```
    std::cout << ymw << '\n';

    ymw += months{1};
    ymw += years{1};

    std::cout << ymw << '\n';

    return 0;
}
```

Output

```
1997/Jan/Wed[1]
1998/Feb/Wed[1]
```

`operator-=`

Subtract months or years from this `year_month_weekday`.

C++

```
1) constexpr year_month_weekday& operator-=(const months& m) noexcept;
2) constexpr year_month_weekday& operator-=(const years& y) noexcept;
```

Parameters

m

The number of months to subtract.

y

The number of years to subtract.

Return value

`*this` which reflects the result of the subtraction.

Example: `operator-=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>
```

```
using namespace std::chrono;

int main()
{
    year_month_weekday ymw{1997y / January / Wednesday[1]};
    std::cout << ymw << '\n';

    ymw -= months{1};
    ymw -= years{1};

    std::cout << ymw << '\n';

    return 0;
}
```

Output

```
1997/Jan/Wed[1]
1995/Dec/Wed[1]
```

`operator local_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_weekday` as local_days

C++

```
constexpr explicit operator local_days() const noexcept;
```

Return value

If `ok()`, returns a count of days as `local_days{sys_days{*this}.time_since_epoch()}`. Otherwise the returned value is unspecified.

`operator sys_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_day` as sys_days.

C++

```
constexpr operator sys_days() const noexcept;
```

Return value

If `ok()`, returns a `sys_days` that represents the date that is `(index() - 1) * 7` days after the first `weekday()` of `year()/month()`. If `index()` is `0`, the returned `sys_days` represents the date 7 days before the first `weekday()` of `year()/month()`.

`weekday`

Get the weekday stored in the weekday_indexed stored in this `year_month_weekday`.

C++

```
constexpr weekday weekday() const noexcept;
```

Return value

The weekday value.

`weekday_indexed`

Get the weekday_indexed stored in this `year_month_weekday`.

C++

```
constexpr weekday_indexed weekday_indexed() const noexcept;
```

Return value

The weekday_indexed value.

`year`

Get the year value.

C++

```
constexpr year year() const noexcept;
```

Return value

The year value.

See also

<chrono>
year
year_month
year_month_day
year_month_day_last
year_month_weekday_last
operator/ Header Files Reference

`year_month_weekday_last` class

Article • 06/15/2022

A specific year, month, and last weekday of the month.

Syntax

C++

```
class year_month_weekday_last; // C++ 20
```

Remarks

`year_month_weekday_last` supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For days-oriented arithmetic, use the `sys_days` conversion to convert to a sys_days, which supports days-oriented arithmetic.

`year_month_weekday_last` is a trivially copyable and standard-layout class type.

Members

Name	Description
Constructor	Construct a `year_month_weekday_last` with the specified month and weekday.
month	Get the month value.
ok	Check if the `year_month_weekday_last` is valid.
operator+=	Add the specified number of months or years.
operator-=	Subtract the specified number of months or years.
operator local_days	Get the count of days from the `system_clock` epoch to this `year_month_weekday_last` as local_days.
operator sys_days	Get the count of days from the `system_clock` epoch to this `year_month_weekday_last` as sys_days.
weekday	Get the weekday.
weekday_last	Get the `weekday_last` stored in this `year_month_weekday_last`.

Name	Description
year	Get the year.

Non-members

Name	Description
operator+	Add months or years.
operator-	Subtract months or years.
operator==	Determine whether two `year_month_weekday_last` values are equal.
operator<<	Output a `year_month_weekday_last` to the given stream.

Requirements

Header: `<chrono>` (since C++20)

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructor

Constructs a `year_month_weekday_last`.

```
C++
```

```
constexpr year_month_weekday_last(const year& y, const month& m, const
weekday_last& wdl) noexcept
```

Parameters

`m`

The month value.

`wdl`

The weekday_last value.

`y`

The year value.

For information about C++ 20 syntax used to specify dates, see operator/

Example: Create a `year_month_weekday_last`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_weekday_last ymwl{ 1997y / January / Wednesday[last] };
    std::cout << ymwl << '\n';

    return 0;
}
```

Output

```
1997/Jan/Wed[last]
```

`month`

Get the month value.

C++

```
constexpr month month() const noexcept;
```

Return value

The month value.

`ok`

Check if the value stored in this `year_month_weekday_last` is valid. The `year`, `month`, and `weekday_last` stored in this `year_month_weekday_last` must all be `ok` for this function to

return `true`. Otherwise, returns `false`.

```
C++

constexpr bool ok() const noexcept;
```

Return value

`true` if the `year_month_weekday_last` value is valid. Otherwise, `false`.

A `year_month_weekday_last` is valid if the `month`, `weekday_indexed`, and `year` are all valid.

`operator+=`

Add months or years to this `year_month_weekday_last`.

```
C++

1) constexpr year_month_weekday_last& operator+=(const months& m) noexcept;
2) constexpr year_month_weekday_last& operator+=(const years& y) noexcept;
```

Parameters

`m`

The number of months to add.

`y`

The number of years to add.

Return value

`*this` which reflects the result of the addition.

Example: `operator+=`

```
C++

// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;
```

```
int main()
{
    year_month_weekday_last ymwl{ year(1997) / January / Wednesday[last] };
    std::cout << ymwl << '\n';

    ymwl += months{ 1 };
    ymwl += years{ 1 };

    std::cout << ymwl << '\n';

    return 0;
}
```

Output

```
1997/Jan/Wed[last]
1998/Feb/Wed[last]
```

`operator-=`

Subtract months or years from this `year_month_weekday_last`.

C++

```
1) constexpr year_month_weekday_last& operator-=(const months& m) noexcept;
2) constexpr year_month_weekday_last& operator-=(const years& y) noexcept;
```

Parameters

m

The number of months to subtract.

y

The number of years to subtract.

Return value

`*this` which reflects the result of the subtraction.

Example: `operator-=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    year_month_weekday_last ymwl{ year(1997) / January / Wednesday[last] };
    std::cout << ymwl << '\n';

    ymwl -= months{ 1 };
    ymwl -= years{ 1 };

    std::cout << ymwl << '\n';

    return 0;
}
```

Output

```
1997/Jan/Wed[last]
1995/Dec/Wed[last]
```

`operator local_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_weekday_last` as local_days

C++

```
constexpr explicit operator local_days() const noexcept;
```

Return value

If `ok()`, returns a count of days as `local_days{sys_days{*this}.time_since_epoch()}`. Otherwise the returned value is unspecified.

`operator sys_days`

Get the count of days from the `system_clock` epoch (1/1/1970) to this `year_month_day` as sys_days.

```
C++
```

```
constexpr operator sys_days() const noexcept;
```

Return value

If `ok() == true`, returns a `sys_days` that represents the last `weekday()` of `year() / month()` (note: the `/` is the date operator, not a division). Otherwise the returned value is unspecified.

`weekday`

Get the weekday.

```
C++
```

```
constexpr weekday weekday() const noexcept;
```

Return value

The weekday.

`weekday_last`

Get the weekday_last stored in this `year_month_weekday_last`.

```
C++
```

```
constexpr weekday_indexed weekday_last() const noexcept;
```

Return value

The weekday_last.

`year`

Get the year value.

```
C++
```

```
constexpr year year() const noexcept;
```

Return value

The year value.

See also

<chrono>
year
year_month
year_month_day
year_month_day_last
year_month_weekday
operator/
Header Files Reference

`zoned_time` class

Article • 06/15/2022

A `zoned_time` is a pairing of a time_zone and a time_point. It gives a `time_point` meaning within a specific time zone.

Syntax

```
C++
```

```
template<class Duration, class TimeZonePtr = const time_zone*>
class zoned_time ;  // C++20
```

Remarks

A `zoned_time` always refers to a valid time zone and represents a point in time that exists and isn't ambiguous in that time zone.

Members

Name	Description
Constructors	Construct a `zoned_time`.
get_info	Uses the time zone in this `zoned_time` to get information about the time point in the `zoned_time`.
get_local_time	Combines the time zone in the `zoned_time` with the stored time point to produce the local time in that time zone.
get_sys_time	Gets the time point stored in this `zoned_time` without applying the time zone.
get_time_zone	Gets the time_zone stored in this `zoned_time`.
operator=	Assign the value of another `zoned_time`, or another `zoned_time`'s `local_time` or `sys_time`, to this `zoned_time`.
operator local_time	Combines the `zoned_time`'s time zone with the stored time point to produce the local time in that time zone.
operator sys_time	Gets the time stored in this `zoned_time` without applying the time zone.

Non-members

Name	Description
operator==	Determine whether two `zoned_time` instances are equal.
operator<<	Output a `zoned_time` to the given stream.

Convenience type aliases

Name	Description
`zoned_seconds`	A synonym for `zoned_time<seconds>;`
`zoned_time::duration`	A duration measured in seconds. It's a synonym for `common_type_t<Duration, seconds>;`

Requirements

Header: `<chrono>`

Microsoft C++ supports the `zoned_time` class starting in Visual Studio 2019 version 16.10.

Time-zone data is only available for Windows 10 version 1903/19H1 and later, and Windows Server 2022 and later.

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Constructors

Construct a `zoned_time`.

C++

```
1) zoned_time();
2) zoned_time(const zoned_time& zt) = default;
3) explicit zoned_time(TimeZonePtr z);
4) zoned_time(const sys_time<Duration>& st);
5) explicit zoned_time(string_view name);
6) zoned_time(TimeZonePtr z, const local_time<Duration>& tp);
7) zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);
```

```
8) zoned_time(TimeZonePtr z, const sys_time<Duration>& st);
9) zoned_time(string_view name, const local_time<Duration>& tp);
10) zoned_time(string_view name, const local_time<Duration>& tp, choose c);
11) zoned_time(string_view name, const sys_time<Duration>& st);
12) template<class Duration, class TimeZonePtr> zoned_time(string_view name,
const zoned_time<Duration>& y);
13) template<class Duration, class TimeZonePtr> zoned_time(string_view name,
const zoned_time<Duration>& y, choose c);
14) template<class Duration> zoned_time(const zoned_time<Duration,
TimeZonePtr>& y);
15) template<class Duration, class TimeZonePtr> zoned_time(TimeZonePtr z,
const zoned_time<Duration, TimeZonePtr>& y);
16) template<class Duration, class TimeZonePtr> zoned_time(TimeZonePtr z,
const zoned_time<Duration, TimeZonePtr>& y, choose);
```

Parameters

c

Indicates how to handle ambiguous or nonexistent local times when converting a `local_time` to a `sys_time`. For more information, see choose enum.

name

The name of a time zone.

st

A `sys_time` with a specified duration. Used to set the time.

tp

A `local_time` with a specified duration. Used to set the time.

y

A `zoned_time` copied to construct a new `zoned_time`.

z

A time_zone that is `std::move(z)`'d into the constructed `zoned_time`.

zt

A `zoned_time` pointer that is `std::move(zt)`'d into the constructed `zoned_time`.

Remarks

1) Initializes the time zone with `traits::default_zone()`, and default constructs the time point.
2) The default copy constructor.

3) Initializes the time zone with `std::move(z)`, and default constructs the time point.
4) Initializes the time zone with `traits::default_zone()`, and the time with `st`.
5) Initializes the time zone with `traits::locate_zone(name)` and default constructs the time point.
6) Initializes the time zone with `std::move(z)`, and the time by converting `tp` as though by `z->to_sys(tp)`.
7) Initializes the time zone with `std::move(z)`, and the time as though by `z->to_sys(tp, c)`. See choose enum for how the parameter `c` affects the outcome.
8) Initializes the time zone with `std::move(z)`, and the time with `st`.
9) Equivalent to construction with `{traits::locate_zone(name), tp}`.
10) Equivalent to construction with `{traits::locate_zone(name), tp, c}`.
11) Equivalent to construction with `{traits::locate_zone(name), st}`.
12) Equivalent to construction with `{traits::locate_zone(name), y}`.
13) Equivalent to construction with `{traits::locate_zone(name), y, c}`. The `choose` parameter, `c`, has no effect.
14) Initializes the time zone from `y`'s time zone and time point.
15) Initializes the time zone with `std::move(z)` and the time from `y`'s time point.
16) Equivalent to construction with `{z, y}`. The `choose` parameter, `c`, has no effect.

> ⓘ Note
>
> `zoned_time` doesn't have a move constructor. Trying to move it results in a copy using the default copy constructor.

Example: construct a `zoned_time`

The following shows how to create a `zoned_time` instance for the time zone `"Antartica/Casey"`, on 9/15/2021 at 4:45pm:

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Antarctica/Casey",
sys_days{2021y/September/15d}+16h+45min);
    std::cout << zt;
```

```
    return 0;
}
```

Output

```
2021-09-16 03:45:00 GMT+11
```

`get_info`

Uses the time zone in this `zoned_time` to get information about the time point in the `zoned_time`.

C++

```
sys_info get_info() const;
```

Return value

A sys_info that contains the UTC offset for the time zone and time, the daylight savings time adjustment offset, and so on.

Example: `get_info`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Antarctica/Casey",
sys_days{2021y/September/15d}+16h+45min);
    sys_info si = zt.get_info();
    std::cout << si;

    return 0;
}
```

Output

```
begin: 2020-10-03 16:01:00, end: 32767-12-31 23:59:59, offset: 39600s, save:
0min, abbrev: GMT+11
```

`get_local_time`

Gets a `local_time<duration>` that represents the local time given this `zoned_time`'s time zone and time point.

C++

```
local_time<duration> get_local_time() const;
```

Return value

The returned `local_time` is a synonym for `time_point<local_t, duration>`.

Example: `get_local_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Antarctica/Casey",
sys_days{2021y/September/15d}+16h+45min);
    std::cout << zt.get_info();

    return 0;
}
```

Output

```
2021-09-16 03:45:00
```

`get_sys_time`

Gets the time stored in the `zoned_time` for the system_clock.

C++

```
sys_time<duration> get_sys_time() const;
```

Return value

The time stored in this `zoned_time`, without any adjustment for time zone. It represents the system-wide wall clock time as a time_point. The returned `sys_time` is a synonym for `template <class Duration> time_point<system_clock Duration>;`

Example: `get_sys_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Antarctica/Casey",
sys_days{2021y/September/15d}+16h+45min);
    std::cout << zt.get_sys_time();

    return 0;
}
```

Output

```
2021-09-15 16:45:00
```

`get_time_zone`

Gets the time zone stored in this `zoned_time`.

C++

```
TimeZonePtr get_time_zone() const;
```

Return value

A pointer to the time_zone stored in this `zoned_time`.

Example: `get_time_zone`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Antarctica/Casey",
sys_days{2021y/September/15d}+16h+45min);
    auto pTZ = zt.get_time_zone();
    std::cout << pTZ->name();

    return 0;
}
```

Output

```
Antarctica/Casey
```

`operator=`

Assign another `zoned_time` to `this` one. Or assign just the other `zoned_time`'s `sys_time` or `local_time`.

C++

```
1) zoned_time& operator=(const zoned_time& zt) = default;
2) zoned_time& operator=(const sys_time<Duration>& st);
3) zoned_time& operator=(const local_time<Duration>& lt);
```

Parameters

`zt`

The `zoned_time` to assign to `this` one.

`st`

The `sys_time` to assign to the stored `time_point` in this `zoned_time`.

lt

The `local_time` to assign to the stored `time_point` in this `zoned_time`.

Return value

`*this`

Example: `operator=`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt1{}, zt2("Pacific/Galapagos", sys_days{2021y/September/15d}
+ 16h + 45min);
    std::cout << zt2.get_local_time() << "\n";
    zt1 = zt2.get_local_time();
    std::cout << zt1.get_local_time();

    return 0;
}
```

Output

```
2021-09-15 10:45:00
2021-09-15 10:45:00
```

Remarks

1) The default copy assignment operator. Copies (doesn't move) the stored time_point and time_zone pointer from the other `zoned_time` into this `zoned_time`. 2) Assigns `st` to the time_point in this `zoned_time`. After the assignment, `*this->get_sys_time() == st;` 3) Converts `lt` (a `local_time`) to a `sys_time`. It does this essentially as `timeZone->to_sys(lt)`, and assigns the result to the [`time_point`] in this `zoned_time`. After the assignment, `*this->get_local_time() == lt;`

`operator local_time`

Combines the `zoned_time`'s time zone with the stored time point to produce the local time in that time zone.

C++

```
explicit operator local_time<duration>() const;
```

Return value

A `time_point<local_t, Duration>` that represents the local time for the time and time zone in this `zoned_time`. You can use the convenience alias `local_time` to represent a value returned by this function.

The return value is effectively `timeZone->to_local(get_sys_time());`

Example: `operator local_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Pacific/Galapagos", sys_days{ 2021y / September / 15d } +
16h + 45min);
    std::cout << (local_time<seconds>)zt;

    return 0;
}
```

Output

```
2021-09-15 10:45:00
```

`operator sys_time`

Gets the time point stored in this `zoned_time` without applying the time zone.

C++

```
operator sys_time<duration>() const;
```

Return value

The time stored in this `zoned_time`, without any adjustment for time zone. The returned `sys_time` is a synonym for `template <class Duration> time_point<system_clock Duration>;` and represents the same point in time that is stored in this `zoned_time`.

Example: `operator sys_time`

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

int main()
{
    zoned_time zt("Pacific/Galapagos", sys_days{2021y/September/15d} + 16h +
45min);
    std::cout << (sys_time<seconds>)zt;

    return 0;
}
```

Output

```
2021-09-15 16:45:00
```

See also

<chrono>
time_point
time_zone
zoned_traits struct
Header files reference

`zoned_traits` struct

Article • 06/15/2022

Allows you to associate a different default time zone with a `zoned_time`, and optionally map a custom name to a time zone.

Syntax

```
C++
```

```
1)
template<class T>
struct zoned_traits {}; // C++20

2)
template <>
struct zoned_traits<const time_zone*>; // C++20
```

Parameters

1) `T` - A pointer to a type that provides a custom `time_zone`.
2) When you don't provide the `time_zone*` template argument, this specialization provides `const std::chrono::time_zone*`, which defaults to the UTC time zone.

Remarks

The pointer to the type that you supply doesn't have to provide the static functions `default_zone()` or `locate_zone()`. But if it doesn't, the `zoned_time` constructors won't be considered during overload resolution.

Members

Name	Description
default_zone	Gets the `time_zone` pointer for the default time zone.
locate_zone	Gets the `time_zone` pointer for the specified time zone.

Requirements

Header: `<chrono>`

Namespace: `std::chrono`

Compiler Option: /std:c++latest

Microsoft C++ supports `zoned_traits` class starting in Visual Studio 2019 version 16.10.

Time-zone data is only available for Windows 10 version 1903/19H1 and later, and Windows Server 2022 and later.

`default_zone`

Gets the time_zone for the default time zone. For more information about how this works, see the code example at the end of this topic.

```
C++
```

```
static const time_zone* default_zone();
```

Return value

If the template argument isn't provided, then the template specialization provides `zoned_traits<const time_zone*>`, which returns the UTC time zone. Otherwise, it returns the default time zone provided by the template argument `T`.

`locate_zone`

Returns the time_zone pointer for the specified time zone. For more information about how this works, see the code example at the end of this topic.

```
C++
```

```
static const time_zone* locate_zone(string_view name);
```

Template parameters

`name`

The name of the time zone to locate. For example, `"UTC"`.

Return value

If you constructed a `zoned_traits` without providing the template argument of a custom time zone pointer, then the return value is `std::chrono::locate_zone(name)`. Otherwise, it returns the value of `locate_zone()` as defined in the template argument `T`.

Example: `zoned_traits`

This following example shows how to use `zoned_traits` to supply a custom default time zone.

First, `CustomTimeZonePtr` is defined, which provides the pointer to the custom time zone type via `operator->()`.

Then, a `zoned_traits` is declared in which `default_zone` is defined to return the custom default time zone. In this case, the South Pole.

In the example, `zoned_traits<CustomTimeZonePtr>::locate_zone()` passes the specified time zone name to `std::chrono::locate_zone()`. This function is where you could map a custom time zone name to another time zone.

Finally, `stdZT` is defined, which uses the standard `time_zone` pointer because it doesn't provide a template argument, so the specialization is used that provides `const std::chrono::time_zone*`.

Run this example to see the `zoned_time` use the custom, and then the standard, `time_zone` pointer.

C++

```
// compile using: /std:c++latest
#include <iostream>
#include <chrono>

using namespace std::chrono;

struct CustomTimeZonePtr
{
    CustomTimeZonePtr() {}
    CustomTimeZonePtr(const time_zone* tz) : tzptr(tz) {}

    const time_zone* operator->() const
    {
        return tzptr;
    }
```

```
private:
    const time_zone* tzptr;
};

template <>
struct zoned_traits<CustomTimeZonePtr>
{
    static const CustomTimeZonePtr default_zone()
    {
        return CustomTimeZonePtr{ locate_zone("Antarctica/South_Pole") };
    }

    static const CustomTimeZonePtr locate_zone(std::string_view name)
    {
        // Here you can provide your own mapping between the name
        // parameter and another time zone before passing it to
locate_zone()
        return CustomTimeZonePtr{ std::chrono::locate_zone(name) };
    }
};

int main()
{
    std::cout << "-- Custom time zone pointer with specialized zoned_traits
and different default zone behavior --\n";
    zoned_time<seconds, CustomTimeZonePtr> myZT;
    std::cout << myZT.get_info() << "\n";

    std::cout << "-- Built-in time zone pointer with standard zoned_traits -
-\n";
    zoned_time<seconds, const time_zone*> stdZT;
    std::cout << stdZT.get_info() << "\n";

    return 0;
}
```

Output

```
-- Custom time zone pointer with specialized zoned_traits and different
default zone behavior --
begin: 1945-12-31 12:00:00, end: 1974-11-02 14:00:00, offset: 43200s, save:
0min, abbrev: GMT+12
-- Built-in time zone pointer with standard zoned_traits --
begin: -32767-01-01 00:00:00, end: 32767-12-31 23:59:59, offset: 0s, save:
0min, abbrev: UTC
```

See also

<chrono>

time_zone

zoned_time class
Header files reference

<cinttypes>

Article • 12/07/2021

Includes the Standard C library header <inttypes.h>. Including this header also includes <cstdint>.

Syntax

```
C++

#include <cinttypes>
```

Remarks

Including this header ensures that the names declared by using external linkage in the Standard C library header are declared in the `std` namespace.

See also

<cstdint>
Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

`<ciso646>`

Article • 12/07/2021

Includes the C standard library header <iso646.h>, and adds the associated names to the `std` namespace.

> **Note**
>
> Because this compatibility header defines names that are keywords in C++, including it has no effect. The <iso646.h> header is deprecated in C++. The <ciso646> header is removed in the draft C++20 standard.

Requirements

Header: <ciso646>

Namespace: std

Remarks

Including this header ensures that the names declared using external linkage in the C standard library header are declared in the `std` namespace.

See also

Header files reference
C++ standard library overview
Thread safety in the C++ standard library

<climits>

Article • 12/07/2021

Includes the C standard library header <limits.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C standard library header are declared in the `std` namespace.

Syntax

C++

```
#include <climits>
```

Macros

C++

```
#define CHAR_BIT
#define SCHAR_MIN
#define SCHAR_MAX
#define UCHAR_MAX
#define CHAR_MIN
#define CHAR_MAX
#define MB_LEN_MAX
#define SHRT_MIN
#define SHRT_MAX
#define USHRT_MAX
#define INT_MIN
#define INT_MAX
#define UINT_MAX
#define LONG_MIN
#define LONG_MAX
#define ULONG_MAX
#define LLONG_MIN
#define LLONG_MAX
#define ULLONG_MAX
```

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<clocale>

Article • 12/07/2021

Includes the Standard C library header <locale.h> and adds the associated names to the `std` namespace.

Requirements

Header: <clocale>

Namespace: std

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants

C++

```
#define NULL see below
#define LC_ALL see below
#define LC_COLLATE see below
#define LC_CTYPE see below
#define LC_MONETARY see below
#define LC_NUMERIC see below
#define LC_TIME see below
```

Structures

C++

```
struct lconv;
```

Functions

C++

```
char* setlocale(int category, const char* locale);
lconv* localeconv();
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cmath>

Article • 01/25/2022

Includes the Standard C library header `<math.h>` and adds the associated names to the `std` namespace.

Syntax

C++

```
#include <cmath>
```

Constants and Types

C++

```
namespace std {
    using float_t = see below ;
    using double_t = see below ;
}

#define HUGE_VAL see below
#define HUGE_VALF see below
#define HUGE_VALL see below
#define INFINITY see below
#define NAN see below
#define FP_INFINITE see below
#define FP_NAN see below
#define FP_NORMAL see below
#define FP_SUBNORMAL see below
#define FP_ZERO see below
#define FP_FAST_FMA see below
#define FP_FAST_FMAF see below
#define FP_FAST_FMAL see below
#define FP_ILOGB0 see below
#define FP_ILOGBNAN see below
#define MATH_ERRNO see below
#define MATH_ERREXCEPT see below
#define math_errhandling see below
```

Functions

C++

```
float acos(float x);
double acos(double x);
long double acos(long double x);
float acosf(float x);
long double acosl(long double x);
float asin(float x);
double asin(double x);
long double asin(long double x);
float asinf(float x);
long double asinl(long double x);
float atan(float x);
double atan(double x);
long double atan(long double x);
float atanf(float x);
long double atanl(long double x);
float atan2(float y, float x);
double atan2(double y, double x);
long double atan2(long double y, long double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
float cos(float x);
double cos(double x);
long double cos(long double x);
float cosf(float x);
long double cosl(long double x);
float sin(float x);
double sin(double x);
long double sin(long double x);
float sinf(float x);
long double sinl(long double x);
float tan(float x);
double tan(double x);
long double tan(long double x);
float tanf(float x);
long double tanl(long double x);
float acosh(float x);
double acosh(double x);
long double acosh(long double x);
float acoshf(float x);
long double acoshl(long double x);
float asinh(float x);
double asinh(double x);
long double asinh(long double x);
float asinhf(float x);
long double asinhl(long double x);
float atanh(float x);
double atanh(double x);
long double atanh(long double x);
float atanhf(float x);
long double atanhl(long double x);
float cosh(float x);
double cosh(double x);
long double cosh(long double x);
float coshf(float x);
long double coshl(long double x);
```

```
float sinh(float x);
double sinh(double x);
long double sinh(long double x);
float sinhf(float x);
long double sinhl(long double x);
float tanh(float x);
double tanh(double x);
long double tanh(long double x);
float tanhf(float x);
long double tanhl(long double x);
float exp(float x);
double exp(double x);
long double exp(long double x);
float expf(float x);
long double expl(long double x);
float exp2(float x);
double exp2(double x);
long double exp2(long double x);
float exp2f(float x);
long double exp2l(long double x);
float expm1(float x);
double expm1(double x);
long double expm1(long double x);
float expm1f(float x);
long double expm1l(long double x);
float frexp(float value, int* exp);
double frexp(double value, int* exp);
long double frexp(long double value, int* exp);
float frexpf(float value, int* exp);
long double frexpl(long double value, int* exp);
int ilogb(float x);
int ilogb(double x);
int ilogb(long double x);
int ilogbf(float x);
int ilogbl(long double x);
float ldexp(float x, int exp);
double ldexp(double x, int exp);
long double ldexp(long double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
float log(float x);
double log(double x);
long double log(long double x);
float logf(float x);
long double logl(long double x);
float log10(float x);
double log10(double x);
long double log10(long double x);
float log10f(float x);
long double log10l(long double x);
float log1p(float x);
double log1p(double x);
long double log1p(long double x);
float log1pf(float x);
long double log1pl(long double x);
```

```
float log2(float x);
double log2(double x);
long double log2(long double x);
float log2f(float x);
long double log2l(long double x);
float logb(float x);
double logb(double x);
long double logb(long double x);
float logbf(float x);
long double logbl(long double x);
float modf(float value, float* iptr);
double modf(double value, double* iptr);
long double modf(long double value, long double* iptr);
float modff(float value, float* iptr);
long double modfl(long double value, long double* iptr);
float scalbn(float x, int n);
double scalbn(double x, int n);
long double scalbn(long double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
float scalbln(float x, long int n);
double scalbln(double x, long int n);
long double scalbln(long double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
float cbrt(float x);
double cbrt(double x);
long double cbrt(long double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Absolute values

C++

```
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);
float fabs(float x);
double fabs(double x);
long double fabs(long double x);
float fabsf(float x);
long double fabsl(long double x);
float hypot(float x, float y);
double hypot(double x, double y);
long double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Three-dimensional hypotenuse

C++

```
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);
float pow(float x, float y);
double pow(double x, double y);
long double pow(long double x, long double y);
float powf(float x, float y);
long double powl(long double x, long double y);
float sqrt(float x);
double sqrt(double x);
long double sqrt(long double x);
float sqrtf(float x);
long double sqrtl(long double x);
float erf(float x);
double erf(double x);
long double erf(long double x);
float erff(float x);
long double erfl(long double x);
float erfc(float x);
double erfc(double x);
long double erfc(long double x);
float erfcf(float x);
long double erfcl(long double x);
float lgamma(float x);
double lgamma(double x);
long double lgamma(long double x);
float lgammaf(float x);
long double lgammal(long double x);
float tgamma(float x);
double tgamma(double x);
long double tgamma(long double x);
float tgammaf(float x);
long double tgammal(long double x);
float ceil(float x);
double ceil(double x);
long double ceil(long double x);
float ceilf(float x);
long double ceill(long double x);
float floor(float x);
double floor(double x);
long double floor(long double x);
float floorf(float x);
long double floorl(long double x);
float nearbyint(float x);
double nearbyint(double x);
long double nearbyint(long double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
float rint(float x);
```

```
double rint(double x);
long double rint(long double x);
float rintf(float x);
long double rintl(long double x);
long int lrint(float x);
long int lrint(double x);
long int lrint(long double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(float x);
long long int llrint(double x);
long long int llrint(long double x);
long long int llrintf(float x);
long long int llrintl(long double x);
float round(float x);
double round(double x);
long double round(long double x);
float roundf(float x);
long double roundl(long double x);
long int lround(float x);
long int lround(double x);
long int lround(long double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(float x);
long long int llround(double x);
long long int llround(long double x);
long long int llroundf(float x);
long long int llroundl(long double x);
float trunc(float x);
double trunc(double x);
long double trunc(long double x);
float truncf(float x);
long double truncl(long double x);
float fmod(float x, float y);
double fmod(double x, double y);
long double fmod(long double x, long double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
float remainder(float x, float y);
double remainder(double x, double y);
long double remainder(long double x, long double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
float remquo(float x, float y, int* quo);
double remquo(double x, double y, int* quo);
long double remquo(long double x, long double y, int* quo);
float remquof(float x, float y, int* quo);
long double remquol(long double x, long double y, int* quo);
float copysign(float x, float y);
double copysign(double x, double y);
long double copysign(long double x, long double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
double nan(const char* tagp);
```

```
float nanf(const char* tagp);
long double nanl(const char* tagp);
float nextafter(float x, float y);
double nextafter(double x, double y);
long double nextafter(long double x, long double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
float nexttoward(float x, long double y);
double nexttoward(double x, long double y);
long double nexttoward(long double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
float fdim(float x, float y);
double fdim(double x, double y);
long double fdim(long double x, long double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
float fmax(float x, float y);
double fmax(double x, double y);
long double fmax(long double x, long double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
float fmin(float x, float y);
double fmin(double x, double y);
long double fmin(long double x, long double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
float fma(float x, float y, float z);
double fma(double x, double y, double z);
long double fma(long double x, long double y, long double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

Classification / comparison functions

C++

```
int fpclassify(float x);
int fpclassify(double x);
int fpclassify(long double x);
int isfinite(float x);
int isfinite(double x);
int isfinite(long double x);
int isinf(float x);
int isinf(double x);
int isinf(long double x);
int isnan(float x);
int isnan(double x);
int isnan(long double x);
int isnormal(float x);
int isnormal(double x);
int isnormal(long double x);
```

```
int signbit(float x);
int signbit(double x);
int signbit(long double x);
int isgreater(float x, float y);
int isgreater(double x, double y);
int isgreater(long double x, long double y);
int isgreaterequal(float x, float y);
int isgreaterequal(double x, double y);
int isgreaterequal(long double x, long double y);
int isless(float x, float y);
int isless(double x, double y);
int isless(long double x, long double y);
int islessequal(float x, float y);
int islessequal(double x, double y);
int islessequal(long double x, long double y);
int islessgreater(float x, float y);
int islessgreater(double x, double y);
int islessgreater(long double x, long double y);
int isunordered(float x, float y);
int isunordered(double x, double y);
int isunordered(long double x, long double y);
```

Mathematical special functions

Associated Laguerre polynomials

C++

```
double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);
```

Associated Legendre functions

C++

```
double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);
```

Beta function

C++

```
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);
```

Complete elliptic integral of the first kind

C++

```
double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);
```

Complete elliptic integral of the second kind

C++

```
double comp_ellint_2(double k);
float comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);
```

Complete elliptic integral of the third kind

C++

```
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
```

Regular modified cylindrical Bessel functions

C++

```
double cyl_bessel_i(double nu, double x);
float cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);
```

Cylindrical Bessel functions of the first kind

C++

```
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
```

Irregular modified cylindrical Bessel functions

C++

```
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
```

Cylindrical Neumann functions

C++

```
double cyl_neumann(double nu, double x);
float cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);
```

Incomplete elliptic integral of the first kind

C++

```
double ellint_1(double k, double phi);
float ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);
```

Incomplete elliptic integral of the second kind

C++

```
double ellint_2(double k, double phi);
float ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);
```

Incomplete elliptic integral of the third kind

C++

```
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);
```

Exponential integral

C++

```
double expint(double x);
float expintf(float x);
long double expintl(long double x);
```

Hermite polynomials

C++

```
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
```

Laguerre polynomials

C++

```
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);
```

Legendre polynomials

C++

```
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);
```

Riemann zeta function

C++

```
double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);
```

Spherical Bessel functions of the first kind

C++

```
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);
```

Spherical associated Legendre functions

C++

```
double sph_legendre(unsigned l, unsigned m, double theta);
float sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);
```

Spherical Neumann functions

C++

```
double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<codecvt>

Article • 12/07/2021

Defines several class templates that describe objects based on class template codecvt. These objects can serve as locale facets that control conversions between a sequence of values of type `Elem` and a sequence of values of type `char`.

Syntax

```
C++

#include <codecvt>
```

Remarks

The locale facets declared in this header convert between several character encodings. For wide characters (stored within the program in fixed-size integers):

- UCS-4 is Unicode (ISO 10646) encoded within the program as a 32-bit integer.
- UCS-2 is Unicode encoded within the program as a 16-bit integer.
- UTF-16 is Unicode encoded within the program as either one or two 16-bit integers. (Note that this does not meet all the requirements of a valid wide-character encoding for Standard C or Standard C++. Nevertheless it is widely used as such.)

For byte streams (stored in a file, transmitted as a byte sequence, or stored within the program in an array of `char`):

- UTF-8 is Unicode encoded within a byte stream as one or more eight-bit bytes with a deterministic byte order.
- UTF-16LE is Unicode encoded within a byte stream as UTF-16 with each 16-bit integer presented as two eight-bit bytes, less significant byte first.
- UTF-16BE is Unicode encoded within a byte stream as UTF-16 with each 16-bit integer presented as two eight-bit bytes, more significant byte first.

Enumerations

Name	Description
codecvt_mode	Specifies configuration information for locale facets.

Classes

Class	Description
codecvt_utf8	Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4, and a byte stream encoded as UTF-8.
codecvt_utf8_utf16	Represents a locale facet that converts between wide characters encoded as UTF-16 and a byte stream encoded as UTF-8.
codecvt_utf16	Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE or UTF-16BE.

Requirements

Header: <codecvt>

Namespace: std

See also

Header Files Reference

`<codecvt>` enums

Article • 12/07/2021

codecvt_mode Enumeration

Specifies configuration information for locale facets.

```
C++
```

```
enum codecvt_mode {
    consume_header = 4,
    generate_header = 2,
    little_endian = 1
};
```

Remarks

The enumeration defines three constants that supply configuration information to the locale facets declared in <codecvt>. The distinct values are:

- `consume_header`, to consume an initial header sequence when reading a multibyte sequence and determine the endianness of the subsequent multibyte sequence to be read
- `generate_header`, to generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written
- `little_endian`, to generate a multibyte sequence in little-endian order, as opposed to the default big-endian order

These constants can be ORed together in arbitrary combinations.

See also

<codecvt>

codecvt_utf8

Article • 08/03/2021

Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4, and a byte stream encoded as UTF-8.

```
C++
```

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode =
(codecvt_mode)0>
class codecvt_utf8 : public std::codecvt<Elem, char, StateType>
```

Parameters

Elem
The wide-character element type.

Maxcode
The maximum number of characters for the locale facet.

Mode
Configuration information for the locale facet.

Remarks

The byte stream can be written to either a binary file or a text file.

Requirements

Header: <codecvt>

Namespace: std

codecvt_utf8_utf16

Article • 08/03/2021

Represents a locale facet that converts between wide characters encoded as UTF-16 and a byte stream encoded as UTF-8.

```
C++
```

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode =
(codecvt_mode)0>
class codecvt_utf8_utf16 : public _STD codecvt<Elem, char, StateType>
```

Parameters

Elem
The wide-character element type.

Maxcode
The maximum number of characters for the locale facet.

Mode
Configuration information for the locale facet.

Remarks

The byte stream can be written to either a binary file or a text file.

Requirements

Header: <codecvt>

Namespace: std

codecvt_utf16

Article • 08/03/2021

Represents a locale facet that converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE or UTF-16BE.

```
C++
```

```
template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode =
(codecvt_mode)0>
class codecvt_utf16 : public std::codecvt<Elem, char, StateType>
```

Parameters

Elem
The wide-character element type.

Maxcode
The maximum number of characters for the locale facet.

Mode
Configuration information for the locale facet.

Remarks

This class template converts between wide characters encoded as UCS-2 or UCS-4 and a byte stream encoded as UTF-16LE, if Mode & little_endian, or UTF-16BE otherwise.

The byte stream should be written to a binary file; it can be corrupted if written to a text file.

Requirements

Header: <codecvt>

Namespace: std

`<complex>`

Article • 12/06/2021

Defines the container class template `complex` and its supporting templates.

Requirements

Header: <complex>

Namespace: std

Remarks

A *complex number* is an ordered pair of real numbers. In purely geometrical terms, the *complex plane* is the real, two-dimensional plane. The special qualities of the complex plane that distinguish it from the real plane are due to its having an additional algebraic structure. This algebraic structure has two fundamental operations:

- Addition, defined as $(a, b) + (c, d) = (a + c, b + d)$
- Multiplication, defined as $(a, b) * (c, d) = (ac - bd, ad + bc)$

The set of complex numbers with the operations of complex addition and complex multiplication are a *field* in the standard algebraic sense:

- The operations of addition and multiplication are commutative and associative and multiplication distributes over addition exactly as it does with real addition and multiplication on the field of real numbers.
- The complex number (0, 0) is the additive identity and (1, 0) is the multiplicative identity.
- The additive inverse for a complex number (a, b) is $(-a, -b)$, and the multiplicative inverse for all such complex numbers except (0, 0) is

  $(a/(a^2 + b^2), -b/(a^2 + b^2))$

By representing a complex number $z = (a, b)$ in the form $z = a + bi$, where $i^2 = -1$, the rules for the algebra of the set of real numbers can be applied to the set of complex numbers and to their components. For example:

$(1 + 2i) * (2 + 3i) = 1 * (2 + 3i) + 2i * (2 + 3i) = (2 + 3i) + (4i + 6i^2) = (2 - 6) + (3 + 4)i = -4 + 7i$

The system of complex numbers is a field, but it's not an ordered field. There's no ordering of the complex numbers as there is for the field of real numbers and its subsets, so inequalities can't be applied to complex numbers as they are to real numbers.

There are three common forms of representing a complex number z:

- Cartesian: $z = a + bi$
- Polar: $z = r (\cos p + i \sin p)$
- Exponential: $z = r * e^{ip}$

The terms used in these standard representations of a complex number are referred to as follows:

- The real Cartesian component or real part a.
- The imaginary Cartesian component or imaginary part b.
- The modulus or absolute value of a complex number r.
- The argument or phase angle p in radians.

Unless otherwise specified, functions that can return multiple values are required to return a principal value for their arguments greater than -π and less than or equal to +π to keep them single valued. All angles must be expressed in radians, where there are 2π radians (360 degrees) in a circle.

Members

Functions

Name	Description
abs	Calculates the modulus of a complex number.
acos	
acosh	
arg	Extracts the argument from a complex number.

Name	Description
asin	
asinh	
atan	
atanh	
conj	Returns the complex conjugate of a complex number.
cos	Returns the cosine of a complex number.
cosh	Returns the hyperbolic cosine of a complex number.
exp	Returns the exponential function of a complex number.
imag	Extracts the imaginary component of a complex number.
log	Returns the natural logarithm of a complex number.
log10	Returns the base 10 logarithm of a complex number.
norm	Extracts the norm of a complex number.
polar	Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.
pow	Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.
proj	
real	Extracts the real component of a complex number.
sin	Returns the sine of a complex number.
sinh	Returns the hyperbolic sine of a complex number.
sqrt	Returns the square root of a complex number.
tan	Returns the tangent of a complex number.
tanh	Returns the hyperbolic tangent of a complex number.

Operators

Name	Description

Name	Description
operator!=	Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator*	Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator+	Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator-	Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator/	Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator<<	A template function that inserts a complex number into the output stream.
operator==	Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.
operator>>	A template function that extracts a complex value from the input stream.

Classes

Name	Description
complex<double>	The explicitly specialized class template describes an object that stores an ordered pair of objects, both of type `double`, where the first represents the real part of a complex number and the second represents the imaginary part.
complex<float>	The explicitly specialized class template describes an object that stores an ordered pair of objects, both of type `float`, where the first represents the real part of a complex number and the second represents the imaginary part.
complex<long double>	The explicitly specialized class template describes an object that stores an ordered pair of objects, both of type `long double`, where the first represents the real part of a complex number and the second represents the imaginary part.
complex	The class template describes an object used to represent the complex number system and perform complex arithmetic operations.

Literals

The <complex> header defines the following user-defined literals. The literals create a complex number with a real part of zero and an imaginary part that has the value of the

input parameter.

Declaration	Description
`constexpr complex<long double> operator""il(long double d)` `constexpr complex<long double> operator""il(unsigned long long d)`	Returns: `complex<long double>{0.0L, static_cast<long double>(d)}`
`constexpr complex<double> operator""i(long double d)` `constexpr complex<double> operator""i(unsigned long long d)`	Returns: `complex<double>{0.0, static_cast<double>(d)}`.
`constexpr complex<float> operator""if(long double d)` `constexpr complex<float> operator""if(unsigned long long d)`	Returns: `complex<float>{0.0f, static_cast<float>(d)}`.

See also

Header files reference

Thread safety in the C++ standard library

`<complex>` functions

Article • 05/20/2022

abs

Calculates the modulus of a complex number.

C++

```
template <class Type>
Type abs(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose modulus is to be determined.

Return Value

The modulus of a complex number.

Remarks

The *modulus* of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number $a + bi$ is the square root of ($a^2 + b^2$), written $|a + bi|$. The *norm* of a complex number $a + bi$ is ($a^2 + b^2$). The norm of a complex number is the square of its modulus.

Example

C++

```
// complex_abs.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
```

```
   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   // The modulus and argument of a complex number can be recovered
   // using abs & arg member functions
   double absc1 = abs ( c1 );
   double argc1 = arg ( c1 );
   cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
   cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // Testing if the principal angles of c2 and c3 are the same
   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.
```

acos

```
C++
```

```
template<class T> complex<T> acos(const complex<T>&);
```

acosh

```
C++
```

```
template<class T> complex<T> acosh(const complex<T>&);
```

arg

Extracts the argument from a complex number.

```
C++
```

```
template <class Type>
Type arg(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose argument is to be determined.

Return Value

The argument of the complex number.

Remarks

The *argument* is the angle that the complex vector makes with the positive real axis in the complex plane. For a complex number *a* + *bi*, the argument is equal to arctan(*b*/*a*). The angle has a positive sense when measured in a counterclockwise direction from the positive real axis and a negative sense when measured in a clockwise direction. The principal values are greater than -pi and less than or equal to +pi.

Example

C++

```
// complex_arg.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   // The modulus and argument of a complex number can be rcovered
   // using abs & arg member functions
   double absc1 = abs ( c1 );
   double argc1 = arg ( c1 );
   cout << "The modulus of c1 is recovered from c1 using: abs ( c1 ) = "
        << absc1 << endl;
   cout << "Argument of c1 is recovered from c1 using:\n arg ( c1 ) = "
        << argc1 << " radians, which is " << argc1 * 180 / pi
        << " degrees." << endl;

   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // Testing if the principal angles of c2 and c3 are the same
   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The modulus of c1 is recovered from c1 using: abs ( c1 ) = 5
Argument of c1 is recovered from c1 using:
arg ( c1 ) = 0 radians, which is 0 degrees.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The complex numbers c2 & c3 have the same principal arguments.
```

asin

C++

```
template<class T> complex<T> asin(const complex<T>&);
```

asinh

C++

```
template<class T> complex<T> asinh(const complex<T>&);
```

atan

C++

```
template<class T> complex<T> atan(const complex<T>&);
```

atanh

C++

```
template<class T> complex<T> atanh(const complex<T>&);
```

conj

Returns the complex conjugate of a complex number.

C++

```
template <class Type>
complex<Type> conj(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose complex conjugate is being returned.

Return Value

The complex conjugate of the input complex number.

Remarks

The complex conjugate of a complex number *a* + *bi* is *a* - *bi*. The product of a complex number and its conjugate is the norm of the number *a*2 + *b*2.

Example

C++

```
// complex_conj.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;

   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;

   complex <double> c2 = conj ( c1 );
   cout << "The complex conjugate of c1 is c2 = conj ( c1 )= "
        << c2 << endl;
```

```
    double dr2 = real ( c2 );
    cout << "The real part of c2 is real ( c2 ) = "
         << dr2 << "." << endl;

    double di2 = imag ( c2 );
    cout << "The imaginary part of c2 is imag ( c2 ) = "
         << di2 << "." << endl;

    // The real part of the product of a complex number
    // and its conjugate is the norm of the number
    complex <double> c3 = c1 * c2;
    cout << "The norm of (c1 * conj (c1) ) is c1 * c2 = "
         << real( c3 ) << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
The complex conjugate of c1 is c2 = conj ( c1 )= (4,-3)
The real part of c2 is real ( c2 ) = 4.
The imaginary part of c2 is imag ( c2 ) = -3.
The norm of (c1 * conj (c1) ) is c1 * c2 = 25
```

cos

Returns the cosine of a complex number.

C++

```
template <class Type>
complex<Type> cos(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose cosine is being determined.

Return Value

The complex number that is the cosine of the input complex number.

Remarks

Identities defining the complex cosines:

cos (*z*) = (1/2)*(exp (*iz*) + exp (- *iz*))

cos (*z*) = cos (*a* + *bi*) = cos (*a*) cosh (*b*) - isin (*a*) sinh (*b*)

Example

C++

```
// complex_cos.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = cos ( c1 );
   cout << "Complex number c2 = cos ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Cosines of the standard angles in the first
   // two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( cos ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( cos ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( cos ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( cos ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( cos ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( cos ( vc6 ) );

   cout << "The complex components cos (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
```

```
        cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = cos ( c1 ) = (-27.0349,-3.85115)
The modulus of c2 is: 27.3079
The argument of c2 is: -3.00009 radians, which is -171.893 degrees.

The complex components cos (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,1.21529e-013)
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-1.74036e-013)
```

cosh

Returns the hyperbolic cosine of a complex number.

C++

```
template <class Type>
complex<Type> cosh(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose hyperbolic cosine is being determined.

Return Value

The complex number that is the hyperbolic cosine of the input complex number.

Remarks

Identities defining the complex hyperbolic cosines:

cos (*z*) = (1/2)*(exp (*z*) + exp (- *z*))

cos (*z*) = cosh (*a* + *bi*) = cosh (*a*) cos (*b*) + isinh (*a*) sin (*b*)

Example

C++

```
// complex_cosh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = cosh ( c1 );
   cout << "Complex number c2 = cosh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic cosines of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( cosh ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( cosh ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( cosh ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( cosh ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( cosh ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( cosh ( vc6 ) );

   cout << "The complex components cosh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = cosh ( c1 ) = (-6.58066,-7.58155)
The modulus of c2 is: 10.0392
The argument of c2 is: -2.28564 radians, which is -130.957 degrees.

The complex components cosh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(1.22777,0.469075)
(0.730543,0.39695)
(0.540302,-8.70178e-014)
(0.730543,-0.39695)
(1.22777,-0.469075)
(1.54308,2.43059e-013)
```

exp

Returns the exponential function of a complex number.

C++

```
template <class Type>
complex<Type> exp(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose exponential is being determined.

Return Value

The complex number that is the exponential of the input complex number.

Example

C++

```
// complex_exp.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
```

```
   complex <double> c1 ( 1 , pi/6 );
   cout << "Complex number c1 = " << c1 << endl;

   // Value of exponential of a complex number c1:
   // note the argument of c2 is determined by the
   // imaginary part of c1 & the modulus by the real part
   complex <double> c2 = exp ( c1 );
   cout << "Complex number c2 = exp ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Exponentials of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( 0.0 , -pi );
   v1.push_back( exp ( vc1 ) );
   complex <double> vc2  ( 0.0, -2 * pi / 3 );
   v1.push_back( exp ( vc2 ) );
   complex <double> vc3  ( 0.0, 0.0 );
   v1.push_back( exp ( vc3 ) );
   complex <double> vc4  ( 0.0, pi / 3 );
   v1.push_back( exp ( vc4 ) );
   complex <double> vc5  ( 0.0 , 2 * pi / 3 );
   v1.push_back( exp ( vc5 ) );
   complex <double> vc6  ( 0.0, pi );
   v1.push_back( exp ( vc6 ) );

   cout << "The complex components exp (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 3 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
      cout <<  ( * Iter1 ) << "\n     with argument = "
           << ( 180/pi ) * arg ( *Iter1 )
           << " degrees\n     modulus = "
           << abs ( * Iter1 ) << endl;
}
```

imag

Extracts the imaginary component of a complex number.

C++

```
template <class Type>
Type imag(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose real part is to be extracted.

Return Value

The imaginary part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

C++

```
// complexc_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

log

Returns the natural logarithm of a complex number.

C++

```
template <class Type>
complex<Type> log(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose natural logarithm is being determined.

Return Value

The complex number that is the natural logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

C++

```
// complex_log.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of log of a complex number c1
   complex <double> c2 = log ( c1 );
   cout << "Complex number c2 = log ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;
```

```
   // log of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( log ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( log ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( log ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( log ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( log ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( log ( vc6 ) );

   cout << "The complex components log (vci), where abs (vci) = 1 "
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
      cout << *Iter1 << " " << endl;
}
```

log10

Returns the base 10 logarithm of a complex number.

```
C++

template <class Type>
complex<Type> log10(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose base 10 logarithm is being determined.

Return Value

The complex number that is the base 10 logarithm of the input complex number.

Remarks

The branch cuts are along the negative real axis.

Example

C++

```
// complex_log10.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main() {
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of log10 of a complex number c1
   complex <double> c2 = log10 ( c1 );
   cout << "Complex number c2 = log10 ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // log10 of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar (1.0, pi / 6) );
   v1.push_back( log10 ( vc1 ) );
   complex <double> vc2  ( polar (1.0, pi / 3) );
   v1.push_back( log10 ( vc2 ) );
   complex <double> vc3  ( polar (1.0, pi / 2) );
   v1.push_back( log10 ( vc3) );
   complex <double> vc4  ( polar (1.0, 2 * pi / 3) );
   v1.push_back( log10 ( vc4 ) );
   complex <double> vc5  ( polar (1.0, 5 * pi / 6) );
   v1.push_back( log10 ( vc5 ) );
   complex <double> vc6  ( polar (1.0,  pi ) );
   v1.push_back( log10 ( vc6 ) );

   cout << "The complex components log10 (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << endl;
}
```

norm

Extracts the norm of a complex number.

C++

```
template <class Type>
Type norm(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose norm is to be determined.

Return Value

The norm of a complex number.

Remarks

The *norm* of a complex number $a + bi$ is $(a^2 + b^2)$. The norm of a complex number is the square of its modulus. The *modulus* of a complex number is a measure of the length of the vector representing the complex number. The modulus of a complex number $a + bi$ is the square root of $(a^2 + b^2)$, written $|a + bi|$.

Example

C++

```
// complex_norm.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
```

```
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;

   // The modulus and argument of a complex number can be recovered
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // The norm of a complex number is the square of its modulus
   double normc2 = norm ( c2 );
   double sqrtnormc2 = sqrt ( normc2 );
   cout << "The norm of c2 given by: norm ( c2 ) = " << normc2 << endl;
   cout << "The modulus of c2 is the square root of the norm: "
        << "sqrt ( normc2 ) = " << sqrtnormc2 << ".";
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
The norm of c2 given by: norm ( c2 ) = 25
The modulus of c2 is the square root of the norm: sqrt ( normc2 ) = 5.
```

polar

Returns the complex number, which corresponds to a specified modulus and argument, in Cartesian form.

C++

```
template <class Type>
complex<Type> polar(const Type& _Modulus, const Type& _Argument = 0);
```

Parameters

_Modulus
The modulus of the complex number being input.

_Argument
The argument of the complex number being input.

Return Value

Cartesian form of the complex number specified in polar form.

Remarks

The polar form of a complex number provides the modulus *r* and the argument *p*, where these parameters are related to the real and imaginary Cartesian components *a* and *b* by the equations *a* = r * cos *p* and *b* = *r* * sin *p*.

Example

C++

```
// complex_polar.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 5.0 ) );   // Default argument = 0
   complex <double> c2 ( polar ( 5.0 , pi / 6 ) );
   complex <double> c3 ( polar ( 5.0 , 13 * pi / 6 ) );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = polar ( 5.0 , pi / 6 ) = " << c2 << endl;
   cout << "c3 = polar ( 5.0 , 13 * pi / 6 ) = " << c3 << endl;

   if ( (arg ( c2 ) <= ( arg ( c3 ) + .00000001) ) ||
        (arg ( c2 ) >= ( arg ( c3 ) - .00000001) ) )
      cout << "The complex numbers c2 & c3 have the "
           << "same principal arguments."<< endl;
   else
      cout << "The complex numbers c2 & c3 don't have the "
           << "same principal arguments." << endl;
```

```
    // the modulus and argument of a complex number can be rcovered
    double absc2 = abs ( c2 );
    double argc2 = arg ( c2 );
    cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
         << absc2 << endl;
    cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
         << argc2 << " radians, which is " << argc2 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
c1 = polar ( 5.0 ) = (5,0)
c2 = polar ( 5.0 , pi / 6 ) = (4.33013,2.5)
c3 = polar ( 5.0 , 13 * pi / 6 ) = (4.33013,2.5)
The complex numbers c2 & c3 have the same principal arguments.
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.523599 radians, which is 30 degrees.
```

pow

Evaluates the complex number obtained by raising a base that is a complex number to the power of another complex number.

C++

```
template <class Type>
complex<Type> pow(const complex<Type>& _Base, int _Power);

template <class Type>
complex<Type> pow(const complex<Type>& _Base, const Type& _Power);

template <class Type>
complex<Type> pow(const complex<Type>& _Base, const complex<Type>& _Power);

template <class Type>
complex<Type> pow(const Type& _Base, const complex<Type>& _Power);
```

Parameters

_Base

The complex number or number that is of the parameter type for the complex number that is the base to be raised to a power by the member function.

_Power

The integer or complex number or number that is of the parameter type for the complex number that is the power that the base is to be raised to by the member function.

Return Value

The complex number obtained by raising the specified base to the specified power.

Remarks

The functions each effectively convert both operands to the return type, and then return the converted **left** to the power **right**.

The branch cut is along the negative real axis.

Example

C++

```
// complex_pow.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // First member function
   // type complex<double> base & type integer power
   complex <double> cb1 ( 3 , 4);
   int cp1 = 2;
   complex <double> ce1 = pow ( cb1 ,cp1 );

   cout << "Complex number for base cb1 = " << cb1 << endl;
   cout << "Integer for power = " << cp1 << endl;
   cout << "Complex number returned from complex base and integer power:"
        << "\n ce1 = cb1 ^ cp1 = " << ce1 << endl;
   double absce1 = abs ( ce1 );
   double argce1 = arg ( ce1 );
   cout << "The modulus of ce1 is: " << absce1 << endl;
   cout << "The argument of ce1 is: "<< argce1 << " radians, which is "
        << argce1 * 180 / pi << " degrees." << endl << endl;

   // Second member function
   // type complex<double> base & type double power
   complex <double> cb2 ( 3 , 4 );
```

```
   double cp2 = pi;
   complex <double> ce2 = pow ( cb2 ,cp2 );

   cout << "Complex number for base cb2 = " << cb2 << endl;
   cout << "Type double for power cp2 = pi = " << cp2 << endl;
   cout << "Complex number returned from complex base and double power:"
        << "\n ce2 = cb2 ^ cp2 = " << ce2 << endl;
   double absce2 = abs ( ce2 );
   double argce2 = arg ( ce2 );
   cout << "The modulus of ce2 is: " << absce2 << endl;
   cout << "The argument of ce2 is: "<< argce2 << " radians, which is "
        << argce2 * 180 / pi << " degrees." << endl << endl;

   // Third member function
   // type complex<double> base & type complex<double> power
   complex <double> cb3 ( 3 , 4 );
   complex <double> cp3 ( -2 , 1 );
   complex <double> ce3 = pow ( cb3 ,cp3 );

   cout << "Complex number for base cb3 = " << cb3 << endl;
   cout << "Complex number for power cp3= " << cp3 << endl;
   cout << "Complex number returned from complex base and complex power:"
        << "\n ce3 = cb3 ^ cp3 = " << ce3 << endl;
   double absce3 = abs ( ce3 );
   double argce3 = arg ( ce3 );
   cout << "The modulus of ce3 is: " << absce3 << endl;
   cout << "The argument of ce3 is: "<< argce3 << " radians, which is "
        << argce3 * 180 / pi << " degrees." << endl << endl;

   // Fourth member function
   // type double base & type complex<double> power
   double cb4 = pi;
   complex <double> cp4 ( 2 , -1 );
   complex <double> ce4 = pow ( cb4 ,cp4 );

   cout << "Type double for base cb4 = pi = " << cb4 << endl;
   cout << "Complex number for power cp4 = " << cp4 << endl;
   cout << "Complex number returned from double base and complex power:"
        << "\n ce4 = cb4 ^ cp4 = " << ce4 << endl;
   double absce4 = abs ( ce4 );
   double argce4 = arg ( ce4 );
   cout << "The modulus of ce4 is: " << absce4 << endl;
   cout << "The argument of ce4 is: "<< argce4 << " radians, which is "
        << argce4 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
Complex number for base cb1 = (3,4)
Integer for power = 2
Complex number returned from complex base and integer power:
ce1 = cb1 ^ cp1 = (-7,24)
The modulus of ce1 is: 25
```

```
The argument of ce1 is: 1.85459 radians, which is 106.26 degrees.

Complex number for base cb2 = (3,4)
Type double for power cp2 = pi = 3.14159
Complex number returned from complex base and double power:
ce2 = cb2 ^ cp2 = (-152.915,35.5475)
The modulus of ce2 is: 156.993
The argument of ce2 is: 2.91318 radians, which is 166.913 degrees.

Complex number for base cb3 = (3,4)
Complex number for power cp3= (-2,1)
Complex number returned from complex base and complex power:
ce3 = cb3 ^ cp3 = (0.0153517,-0.00384077)
The modulus of ce3 is: 0.0158249
The argument of ce3 is: -0.245153 radians, which is -14.0462 degrees.

Type double for base cb4 = pi = 3.14159
Complex number for power cp4 = (2,-1)
Complex number returned from double base and complex power:
ce4 = cb4 ^ cp4 = (4.07903,-8.98725)
The modulus of ce4 is: 9.8696
The argument of ce4 is: -1.14473 radians, which is -65.5882 degrees.
```

proj

C++

```
template<class T> complex<T> proj(const complex<T>&);
```

real

Extracts the real component of a complex number.

C++

```
template <class Type>
Type real(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose real part is to be extracted.

Return Value

The real part of the complex number as a global function.

Remarks

This template function cannot be used to modify the real part of the complex number. To change the real part, a new complex number must be assigned the component value.

Example

C++

```
// complex_real.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   complex <double> c1 ( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = real ( c1 );
   cout << "The real part of c1 is real ( c1 ) = "
        << dr1 << "." << endl;

   double di1 = imag ( c1 );
   cout << "The imaginary part of c1 is imag ( c1 ) = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is real ( c1 ) = 4.
The imaginary part of c1 is imag ( c1 ) = 3.
```

sin

Returns the sine of a complex number.

C++

```
template <class Type>
complex<Type> sin(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose sine is being determined.

Return Value

The complex number that is the sine of the input complex number.

Remarks

Identities defining the complex sines:

sin (*z*) = (1/2 *i*)*(exp (*iz*) - exp (- *iz*))

sin (*z*) = sin (*a* + *bi*) = sin (*a*) cosh (*b*) + icos (*a*) sinh (*b*)

Example

C++

```
// complex_sin.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of sine of a complex number c1
   complex <double> c2 = sin ( c1 );
   cout << "Complex number c2 = sin ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // sines of the standard angles in the first
   // two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
```

```
   v1.push_back( sin ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( sin ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( sin ( vc3 ) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( sin ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( sin ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( sin ( vc6 ) );

   cout << "The complex components sin (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = sin ( c1 ) = (3.85374,-27.0168)
The modulus of c2 is: 27.2903
The argument of c2 is: -1.42911 radians, which is -81.882 degrees.

The complex components sin (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.85898,0.337596)
(0.670731,0.858637)
(-1.59572e-013,1.1752)
(-0.670731,0.858637)
(-0.85898,0.337596)
(-0.841471,-1.11747e-013)
```

sinh

Returns the hyperbolic sine of a complex number.

C++

```
template <class Type>
complex<Type> sinh(const complex<Type>& complexNum);
```

Parameters

complexNum

The complex number whose hyperbolic sine is being determined.

Return Value

The complex number that is the hyperbolic sine of the input complex number.

Remarks

Identities defining the complex hyperbolic sines:

sinh (*z*) = (1/2)*(exp (*z*) - exp (- *z*))

sinh (*z*) = sinh (*a* + *bi*) = sinh (*a*) cos (*b*) + *i*cosh (*a*) sin (*b*)

Example

C++

```
// complex_sinh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of sine of a complex number c1
   complex <double> c2 = sinh ( c1 );
   cout << "Complex number c2 = sinh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic sines of the standard angles in
   // the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( sinh ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( sinh ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( sinh ( vc3) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( sinh ( vc4 ) );
```

```
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( sinh ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( sinh ( vc6 ) );

   cout << "The complex components sinh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = sinh ( c1 ) = (-6.54812,-7.61923)
The modulus of c2 is: 10.0464
The argument of c2 is: -2.28073 radians, which is -130.676 degrees.

The complex components sinh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.858637,0.670731)
(0.337596,0.85898)
(-5.58735e-014,0.841471)
(-0.337596,0.85898)
(-0.858637,0.670731)
(-1.1752,-3.19145e-013)
```

sqrt

Calculates the square root of a complex number.

C++

```
template <class Type>
complex<Type> sqrt(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose square root is to be found.

Return Value

The square root of a complex number.

Remarks

The square root will have a phase angle in the half-open interval (-pi/2, pi/2].

The branch cuts in the complex plane are along the negative real axis.

The square root of a complex number will have a modulus that is the square root of the input number and an argument that is one-half that of the input number.

Example

C++

```
// complex_sqrt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Complex numbers can be entered in polar form with
   // modulus and argument parameter inputs but are
   // stored in Cartesian form as real & imag coordinates
   complex <double> c1 ( polar ( 25.0 , pi / 2 ) );
   complex <double> c2 = sqrt ( c1 );
   cout << "c1 = polar ( 5.0 ) = " << c1 << endl;
   cout << "c2 = sqrt ( c1 ) = " << c2 << endl;

   // The modulus and argument of a complex number can be recovered
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is recovered from c2 using: abs ( c2 ) = "
        << absc2 << endl;
   cout << "Argument of c2 is recovered from c2 using:\n arg ( c2 ) = "
        << argc2 << " radians, which is " << argc2 * 180 / pi
        << " degrees." << endl;

   // The modulus and argument of c2 can be directly calculated
   absc2 = sqrt( abs ( c1 ) );
   argc2 = 0.5 * arg ( c1 );
   cout << "The modulus of c2 = sqrt( abs ( c1 ) ) =" << absc2 << endl;
   cout << "The argument of c2 = ( 1 / 2 ) * arg ( c1 ) ="
        << argc2 << " radians,\n which is " << argc2 * 180 / pi
        << " degrees." << endl;
}
```

Output

```
c1 = polar ( 5.0 ) = (-2.58529e-012,25)
c2 = sqrt ( c1 ) = (3.53553,3.53553)
The modulus of c2 is recovered from c2 using: abs ( c2 ) = 5
Argument of c2 is recovered from c2 using:
arg ( c2 ) = 0.785398 radians, which is 45 degrees.
The modulus of c2 = sqrt( abs ( c1 ) ) =5
The argument of c2 = ( 1 / 2 ) * arg ( c1 ) =0.785398 radians,
which is 45 degrees.
```

tan

Returns the tangent of a complex number.

C++

```
template <class Type>
complex<Type> tan(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose tangent is being determined.

Return Value

The complex number that is the tangent of the input complex number.

Remarks

Identities defining the complex cotangent:

tan (*z*) = sin (*z*) / cos (*z*) = (exp (*iz*) - exp (- *iz*)) / *i*(exp (*iz*) + exp (- *iz*))

Example

C++

```
// complex_tan.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   // Values of cosine of a complex number c1
   complex <double> c2 = tan ( c1 );
   cout << "Complex number c2 = tan ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic tangent of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( tan ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( tan ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( tan ( vc3) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( tan ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( tan ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0,  pi ) );
   v1.push_back( tan ( vc6 ) );

   cout << "The complex components tan (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
      cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = tan ( c1 ) = (-0.000187346,0.999356)
The modulus of c2 is: 0.999356
The argument of c2 is: 1.57098 radians, which is 90.0107 degrees.

The complex components tan (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.713931,0.85004)
(0.24356,0.792403)
(-4.34302e-014,0.761594)
(-0.24356,0.792403)
```

```
(-0.713931,0.85004)
(-1.55741,-7.08476e-013)
```

tanh

Returns the hyperbolic tangent of a complex number.

```
C++
```

```
template <class Type>
complex<Type> tanh(const complex<Type>& complexNum);
```

Parameters

complexNum
The complex number whose hyperbolic tangent is being determined.

Return Value

The complex number that is the hyperbolic tangent of the input complex number.

Remarks

Identities defining the complex hyperbolic cotangent:

tanh (*z*) = sinh (*z*) / cosh (*z*) = (exp (*z*) - exp (- *z*)) / (exp (*z*) + exp (- *z*))

Example

```
C++
```

```
// complex_tanh.cpp
// compile with: /EHsc
#include <vector>
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   complex <double> c1 ( 3.0 , 4.0 );
   cout << "Complex number c1 = " << c1 << endl;
```

```
   // Values of cosine of a complex number c1
   complex <double> c2 = tanh ( c1 );
   cout << "Complex number c2 = tanh ( c1 ) = " << c2 << endl;
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;

   // Hyperbolic tangents of the standard angles
   // in the first two quadrants of the complex plane
   vector <complex <double> > v1;
   vector <complex <double> >::iterator Iter1;
   complex <double> vc1  ( polar ( 1.0, pi / 6 ) );
   v1.push_back( tanh ( vc1 ) );
   complex <double> vc2  ( polar ( 1.0, pi / 3 ) );
   v1.push_back( tanh ( vc2 ) );
   complex <double> vc3  ( polar ( 1.0, pi / 2 ) );
   v1.push_back( tanh ( vc3 ) );
   complex <double> vc4  ( polar ( 1.0, 2 * pi / 3 ) );
   v1.push_back( tanh ( vc4 ) );
   complex <double> vc5  ( polar ( 1.0, 5 * pi / 6 ) );
   v1.push_back( tanh ( vc5 ) );
   complex <double> vc6  ( polar ( 1.0, pi ) );
   v1.push_back( tanh ( vc6 ) );

   cout << "The complex components tanh (vci), where abs (vci) = 1"
        << "\n& arg (vci) = i * pi / 6 of the vector v1 are:\n" ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = tanh ( c1 ) = (1.00071,0.00490826)
The modulus of c2 is: 1.00072
The argument of c2 is: 0.00490474 radians, which is 0.281021 degrees.

The complex components tanh (vci), where abs (vci) = 1
& arg (vci) = i * pi / 6 of the vector v1 are:
(0.792403,0.24356)
(0.85004,0.713931)
(-3.54238e-013,1.55741)
(-0.85004,0.713931)
(-0.792403,0.24356)
(-0.761594,-8.68604e-014)
```

`<complex>` operators

Article • 12/07/2021

operator!=

Tests for inequality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
C++
```

```
template <class Type>
bool operator!=(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
bool operator!=(
    const complex<Type>& left,
    const Type& right);

template <class Type>
bool operator!=(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left
A complex number or object of its parameter type to be tested for inequality.

right
A complex number or object of its parameter type to be tested for inequality.

Return Value

`true` if the numbers are not equal; `false` if numbers are equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_NE.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> compared with type complex<double>
   complex <double> cl1 ( polar (3.0, pi / 6 ) );
   complex <double> cr1a ( polar (3.0, pi /6 ) );
   complex <double> cr1b ( polar (2.0, pi / 3 ) );

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
   cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
   if ( cl1 != cr1a )
      cout << "The complex numbers cl1 & cr1a are not equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1a are equal." << endl;
   if ( cl1 != cr1b )
      cout << "The complex numbers cl1 & cr1b are not equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1b are equal." << endl;
   cout << endl;

   // Example of the second member function
   // type complex<int> compared with type int
   complex <int> cl2a ( 3, 4 );
   complex <int> cl2b ( 5,0 );
   int cr2a =3;
   int cr2b =5;

   cout << "The 1st left-side complex number is cl2a = " << cl2a << endl;
   cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
   if ( cl2a != cr2a )
      cout << "The complex numbers cl2a & cr2a are not equal." << endl;
   else
      cout << "The complex numbers cl2a & cr2a are equal." << endl;

   cout << "The 2nd left-side complex number is cl2b = " << cl2b << endl;
   cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
```

```
   if ( cl2b != cr2b )
      cout << "The complex numbers cl2b & cr2b are not equal." << endl;
   else
      cout << "The complex numbers cl2b & cr2b are equal." << endl;
   cout << endl;

   // Example of the third member function
   // type double compared with type complex<double>
   double cl3a =3;
   double cl3b =5;
   complex <double> cr3a ( 3, 4 );
   complex <double> cr3b ( 5,0 );

   cout << "The 1st left-side complex number is cl3a = " << cl3a << endl;
   cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
   if ( cl3a != cr3a )
      cout << "The complex numbers cl3a & cr3a are not equal." << endl;
   else
      cout << "The complex numbers cl3a & cr3a are equal." << endl;

   cout << "The 2nd left-side complex number is cl3b = " << cl3b << endl;
   cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
   if ( cl3b != cr3b )
      cout << "The complex numbers cl3b & cr3b are not equal." << endl;
   else
      cout << "The complex numbers cl3b & cr3b are equal." << endl;
   cout << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The 1st right-side complex number is cr1a = (2.59808,1.5)
The 2nd right-side complex number is cr1b = (1,1.73205)
The complex numbers cl1 & cr1a are equal.
The complex numbers cl1 & cr1b are not equal.

The 1st left-side complex number is cl2a = (3,4)
The 1st right-side complex number is cr2a = 3
The complex numbers cl2a & cr2a are not equal.
The 2nd left-side complex number is cl2b = (5,0)
The 2nd right-side complex number is cr2b = 5
The complex numbers cl2b & cr2b are equal.

The 1st left-side complex number is cl3a = 3
The 1st right-side complex number is cr3a = (3,4)
The complex numbers cl3a & cr3a are not equal.
The 2nd left-side complex number is cl3b = 5
The 2nd right-side complex number is cr3b = (5,0)
The complex numbers cl3b & cr3b are equal.
```

operator*

Multiplies two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

```
C++
```

```
template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator*(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left
The first of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

right
The second of two complex numbers or a number that is of the parameter type for a complex number that is to be multiplied by the * operation.

Return Value

The complex number that results from the multiplication of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_mult.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> times type complex<double>
   complex <double> cl1 ( polar (3.0, pi / 6 ) );
   complex <double> cr1 ( polar (2.0, pi / 3 ) );
   complex <double> cs1 = cl1 * cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "Product of two complex numbers is: cs1 = " << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
   cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type complex<double> times type double
   complex <double> cl2 ( polar ( 3.0, pi / 6 ) );
   double cr2 =5;
   complex <double> cs2 = cl2 * cr2;

   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;
   cout << "Product of two complex numbers is: cs2 = " << cs2 << endl;
   double abscs2 = abs ( cs2 );
   double argcs2 = arg ( cs2 );
   cout << "The modulus of cs2 is: " << abscs2 << endl;
   cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

   // Example of the third member function
   // type double times type complex<double>
   double cl3 = 5;
   complex <double> cr3 ( polar (3.0, pi / 6 ) );
   complex <double> cs3 = cl3 * cr3;

   cout << "The left-side complex number is cl3 = " << cl3 << endl;
   cout << "The right-side complex number is cr3 = " << cr3 << endl;
   cout << "Product of two complex numbers is: cs3 = " << cs3 << endl;
   double abscs3 = abs ( cs3 );
   double argcs3 = arg ( cs3 );
   cout << "The modulus of cs3 is: " << abscs3 << endl;
```

```
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
         << argcs3 * 180 / pi << " degrees." << endl << endl;
}
```

operator+

Adds two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

C++

```
template <class Type>
complex<Type> operator+(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator+(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator+(
    const Type& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator+(const complex<Type>& left);
```

Parameters

left
The first of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

right
The second of two complex numbers or a number that is of the parameter type for a complex number that is to be added by the + operation.

Return Value

The complex number that results from the addition of the two numbers whose value and type are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format. The unary operator returns *left*.

Example

C++

```
// complex_op_add.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> plus type complex<double>
   complex <double> cl1 ( 3.0, 4.0 );
   complex <double> cr1 ( 2.0, 5.0 );
   complex <double> cs1 = cl1 + cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "The sum of the two complex numbers is: cs1 = " << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
   cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type complex<double> plus type double
   complex <double> cl2 ( 3.0, 4.0 );
   double cr2 =5.0;
   complex <double> cs2 = cl2 + cr2;

   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;
   cout << "The sum of the two complex numbers is: cs2 = " << cs2 << endl;
   double abscs2 = abs ( cs2 );
   double argcs2 = arg ( cs2 );
   cout << "The modulus of cs2 is: " << abscs2 << endl;
   cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

   // Example of the third member function
```

```
   // type double plus type complex<double>
   double cl3 = 5.0;
   complex <double> cr3 ( 3.0, 4.0 );
   complex <double> cs3 = cl3 + cr3;

   cout << "The left-side complex number is cl3 = " << cl3 << endl;
   cout << "The right-side complex number is cr3 = " << cr3 << endl;
   cout << "The sum of the two complex numbers is: cs3 = " << cs3 << endl;
   double abscs3 = abs ( cs3 );
   double argcs3 = arg ( cs3 );
   cout << "The modulus of cs3 is: " << abscs3 << endl;
   cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;

   // Example of the fourth member function
   // plus type complex<double>
   complex <double> cr4 ( 3.0, 4.0 );
   complex <double> cs4 = + cr4;

   cout << "The right-side complex number is cr4 = " << cr4 << endl;
   cout << "The result of the unary application of + to the right-side"
        << "\n complex number is: cs4 = " << cs4 << endl;
   double abscs4 = abs ( cs4 );
   double argcs4 = arg ( cs4 );
   cout << "The modulus of cs4 is: " << abscs4 << endl;
   cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
        << argcs4 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,5)
The sum of the two complex numbers is: cs1 = (5,9)
The modulus of cs1 is: 10.2956
The argument of cs1 is: 1.0637 radians, which is 60.9454 degrees.

The left-side complex number is cl2 = (3,4)
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = (8,4)
The modulus of cs2 is: 8.94427
The argument of cs2 is: 0.463648 radians, which is 26.5651 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (3,4)
The sum of the two complex numbers is: cs3 = (8,4)
The modulus of cs3 is: 8.94427
The argument of cs3 is: 0.463648 radians, which is 26.5651 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of + to the right-side
complex number is: cs4 = (3,4)
```

```
The modulus of cs4 is: 5
The argument of cs4 is: 0.927295 radians, which is 53.1301 degrees.
```

operator-

Subtracts two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

C++

```
template <class Type>
complex<Type> operator-(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator-(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator-(
    const Type& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator-(const complex<Type>& left);
```

Parameters

left
The first of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

right
The second of two complex numbers or a number that is of the parameter type for a complex number that is to be subtracted by the - operation.

Return Value

The complex number that results from the subtraction of *right* from *left*, the two numbers whose values are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

The unary operator changes the sign of a complex number and returns a value whose real part is the negative of the real part of the number input and whose imaginary part is the negative of the imaginary part of the number input.

Example

C++

```
// complex_op_sub.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> minus type complex<double>
   complex <double> cl1 ( 3.0, 4.0 );
   complex <double> cr1 ( 2.0, 5.0 );
   complex <double> cs1 = cl1 - cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "Difference of two complex numbers is: cs1 = " << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
   cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type complex<double> minus type double
   complex <double> cl2 ( 3.0, 4.0 );
   double cr2 =5.0;
   complex <double> cs2 = cl2 - cr2;

   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;
   cout << "Difference of two complex numbers is: cs2 = " << cs2 << endl;
   double abscs2 = abs ( cs2 );
   double argcs2 = arg ( cs2 );
   cout << "The modulus of cs2 is: " << abscs2 << endl;
   cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;
```

```
   // Example of the third member function
   // type double minus type complex<double>
   double cl3 = 5.0;
   complex <double> cr3 ( 3.0, 4.0 );
   complex <double> cs3 = cl3 - cr3;

   cout << "The left-side complex number is cl3 = " << cl3 << endl;
   cout << "The right-side complex number is cr3 = " << cr3 << endl;
   cout << "Difference of two complex numbers is: cs3 = " << cs3 << endl;
   double abscs3 = abs ( cs3 );
   double argcs3 = arg ( cs3 );
   cout << "The modulus of cs3 is: " << abscs3 << endl;
   cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
        << argcs3 * 180 / pi << " degrees." << endl << endl;

   // Example of the fourth member function
   // minus type complex<double>
   complex <double> cr4 ( 3.0, 4.0 );
   complex <double> cs4 = - cr4;

   cout << "The right-side complex number is cr4 = " << cr4 << endl;
   cout << "The result of the unary application of - to the right-side"
        << "\n complex number is: cs4 = " << cs4 << endl;
   double abscs4 = abs ( cs4 );
   double argcs4 = arg ( cs4 );
   cout << "The modulus of cs4 is: " << abscs4 << endl;
   cout << "The argument of cs4 is: "<< argcs4 << " radians, which is "
        << argcs4 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,5)
Difference of two complex numbers is: cs1 = (1,-1)
The modulus of cs1 is: 1.41421
The argument of cs1 is: -0.785398 radians, which is -45 degrees.

The left-side complex number is cl2 = (3,4)
The right-side complex number is cr2 = 5
Difference of two complex numbers is: cs2 = (-2,4)
The modulus of cs2 is: 4.47214
The argument of cs2 is: 2.03444 radians, which is 116.565 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (3,4)
Difference of two complex numbers is: cs3 = (2,-4)
The modulus of cs3 is: 4.47214
The argument of cs3 is: -1.10715 radians, which is -63.4349 degrees.

The right-side complex number is cr4 = (3,4)
The result of the unary application of - to the right-side
complex number is: cs4 = (-3,-4)
```

```
The modulus of cs4 is: 5
The argument of cs4 is: -2.2143 radians, which is -126.87 degrees.
```

operator/

Divides two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

C++

```
template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
complex<Type> operator*(
    const complex<Type>& left,
    const Type& right);

template <class Type>
complex<Type> operator*(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left
A complex number or a number that is of the parameter type for a complex number that is the numerator to be divided by the denominator with the / operation.

right
A complex number or a number that is of the parameter type for a complex number that is the denominator to be used to divide the numerator with the / operation.

Return Value

The complex number that results from the division of the numerator by the denominator, the values of which are specified by the parameter inputs.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_div.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> divided by type complex<double>
   complex <double> cl1 ( polar ( 3.0, pi / 6 ) );
   complex <double> cr1 ( polar ( 2.0, pi / 3 ) );
   complex <double> cs1 = cl1 / cr1;

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;
   cout << "The quotient of the two complex numbers is: cs1 = cl1 /cr1 = "
        << cs1 << endl;
   double abscs1 = abs ( cs1 );
   double argcs1 = arg ( cs1 );
   cout << "The modulus of cs1 is: " << abscs1 << endl;
   cout << "The argument of cs1 is: "<< argcs1 << " radians, which is "
        << argcs1 * 180 / pi << " degrees." << endl << endl;

   // example of the second member function
   // type complex<double> divided by type double
   complex <double> cl2 ( polar (3.0, pi / 6 ) );
   double cr2 =5;
   complex <double> cs2 = cl2 / cr2;

   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;
   cout << "The quotient of the two complex numbers is: cs2 = cl2 /cr2 = "
        << cs2 << endl;
   double abscs2 = abs ( cs2 );
   double argcs2 = arg ( cs2 );
   cout << "The modulus of cs2 is: " << abscs2 << endl;
   cout << "The argument of cs2 is: "<< argcs2 << " radians, which is "
        << argcs2 * 180 / pi << " degrees." << endl << endl;

   // Example of the third member function
   // type double divided by type complex<double>
   double cl3 = 5;
```

```
    complex <double> cr3 ( polar ( 3.0, pi / 6 ) );
    complex <double> cs3 = cl3 / cr3;

    cout << "The left-side complex number is cl3 = " << cl3 << endl;
    cout << "The right-side complex number is cr3 = " << cr3 << endl;
    cout << "The quotient of the two complex numbers is: cs3 = cl3 /cr2 = "
         << cs3 << endl;
    double abscs3 = abs ( cs3 );
    double argcs3 = arg ( cs3 );
    cout << "The modulus of cs3 is: " << abscs3 << endl;
    cout << "The argument of cs3 is: "<< argcs3 << " radians, which is "
         << argcs3 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
The quotient of the two complex numbers is: cs1 = cl1 /cr1 = (1.29904,-0.75)
The modulus of cs1 is: 1.5
The argument of cs1 is: -0.523599 radians, which is -30 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
The quotient of the two complex numbers is: cs2 = cl2 /cr2 = (0.519615,0.3)
The modulus of cs2 is: 0.6
The argument of cs2 is: 0.523599 radians, which is 30 degrees.

The left-side complex number is cl3 = 5
The right-side complex number is cr3 = (2.59808,1.5)
The quotient of the two complex numbers is: cs3 = cl3 /cr2 =
(1.44338,-0.833333)
The modulus of cs3 is: 1.66667
The argument of cs3 is: -0.523599 radians, which is -30 degrees.
```

operator<<

Inserts a complex number specified into the output stream.

C++

```
template <class Type, class Elem, class Traits>
basic_ostream<Elem, Traits>& operator<<(
    basic_ostream<Elem, Traits>& Ostr,
    const complex<Type>& right);
```

Parameters

Ostr
The output stream into which the complex number is being entered.

right
The complex number to be entered into the output stream

Return Value

Writes the value of the specified complex number to the *Ostr* in a Cartesian format: (*real part, imaginary part*).

Remarks

The output stream is overloaded so that it will accept any form of a complex number, and its default output format is the Cartesian format.

Example

C++

```
// complex_op_insert.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   complex <double> c1 ( 3.0, 4.0 );
   cout << "Complex number c1 = " << c1 << endl;

   complex <double> c2  ( polar ( 2.0, pi / 6 ) );
   cout << "Complex number c2 = " << c2 << endl;

   // To display in polar form
   double absc2 = abs ( c2 );
   double argc2 = arg ( c2 );
   cout << "The modulus of c2 is: " << absc2 << endl;
   cout << "The argument of c2 is: "<< argc2 << " radians, which is "
        << argc2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
Complex number c1 = (3,4)
Complex number c2 = (1.73205,1)
The modulus of c2 is: 2
The argument of c2 is: 0.523599 radians, which is 30 degrees.
```

operator==

Tests for equality between two complex numbers, one or both of which may belong to the subset of the type for the real and imaginary parts.

C++

```
template <class Type>
bool operator==(
    const complex<Type>& left,
    const complex<Type>& right);

template <class Type>
bool operator==(
    const complex<Type>& left,
    const Type& right);

template <class Type>
bool operator==(
    const Type& left,
    const complex<Type>& right);
```

Parameters

left
A complex number or object of its parameter type to be tested for inequality.

right
A complex number or object of its parameter type to be tested for inequality.

Return Value

`true` if the numbers are equal; `false` if numbers are not equal.

Remarks

Two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal. Otherwise, they are unequal.

The operation is overloaded so that comparison tests can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_EQ.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> compared with type complex<double>
   complex <double> cl1 ( polar ( 3.0, pi / 6 ) );
   complex <double> cr1a ( polar ( 3.0, pi /6 ) );
   complex <double> cr1b ( polar ( 2.0, pi / 3 ) );

   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The 1st right-side complex number is cr1a = " << cr1a << endl;
   cout << "The 2nd right-side complex number is cr1b = " << cr1b << endl;
   if ( cl1 == cr1a )
      cout << "The complex numbers cl1 & cr1a are equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1a are not equal." << endl;
   if ( cl1 == cr1b )
      cout << "The complex numbers cl1 & cr1b are equal." << endl;
   else
      cout << "The complex numbers cl1 & cr1b are not equal." << endl;
   cout << endl;

   // Example of the second member function
   // type complex<int> compared with type int
   complex <int> cl2a ( 3, 4 );
   complex <int> cl2b ( 5,0 );
   int cr2a =3;
   int cr2b =5;

   cout << "The 1st left-side complex number is cl2a = " << cl2a << endl;
   cout << "The 1st right-side complex number is cr2a = " << cr2a << endl;
   if ( cl2a == cr2a )
      cout << "The complex numbers cl2a & cr2a are equal." << endl;
   else
      cout << "The complex numbers cl2a & cr2a are not equal." << endl;

   cout << "The 2nd left-side complex number is cl2b = " << cl2b << endl;
   cout << "The 2nd right-side complex number is cr2b = " << cr2b << endl;
```

```
   if ( cl2b == cr2b )
      cout << "The complex numbers cl2b & cr2b are equal." << endl;
   else
      cout << "The complex numbers cl2b & cr2b are not equal." << endl;
   cout << endl;

   // Example of the third member function
   // type double compared with type complex<double>
   double cl3a =3;
   double cl3b =5;
   complex <double> cr3a (3, 4 );
   complex <double> cr3b (5,0 );

   cout << "The 1st left-side complex number is cl3a = " << cl3a << endl;
   cout << "The 1st right-side complex number is cr3a = " << cr3a << endl;
   if ( cl3a == cr3a )
      cout << "The complex numbers cl3a & cr3a are equal." << endl;
   else
      cout << "The complex numbers cl3a & cr3a are not equal." << endl;

   cout << "The 2nd left-side complex number is cl3b = " << cl3b << endl;
   cout << "The 2nd right-side complex number is cr3b = " << cr3b << endl;
   if ( cl3b == cr3b )
      cout << "The complex numbers cl3b & cr3b are equal." << endl;
   else
      cout << "The complex numbers cl3b & cr3b are not equal." << endl;
   cout << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The 1st right-side complex number is cr1a = (2.59808,1.5)
The 2nd right-side complex number is cr1b = (1,1.73205)
The complex numbers cl1 & cr1a are equal.
The complex numbers cl1 & cr1b are not equal.

The 1st left-side complex number is cl2a = (3,4)
The 1st right-side complex number is cr2a = 3
The complex numbers cl2a & cr2a are not equal.
The 2nd left-side complex number is cl2b = (5,0)
The 2nd right-side complex number is cr2b = 5
The complex numbers cl2b & cr2b are equal.

The 1st left-side complex number is cl3a = 3
The 1st right-side complex number is cr3a = (3,4)
The complex numbers cl3a & cr3a are not equal.
The 2nd left-side complex number is cl3b = 5
The 2nd right-side complex number is cr3b = (5,0)
The complex numbers cl3b & cr3b are equal.
```

`operator>>`

Extracts a complex value from the input stream.

```
C++
```

```
template <class Type, class Elem, class Traits>
basic_istream<Elem, Traits>& operator>>(
   basic_istream<Elem, Traits>& Istr,
   complex<Type>& right);
```

Parameters

Istr
The input stream from which the complex number is being extracted.

right
The complex number that is being extracted from the input stream.

Return Value

Reads the value of the specified complex number from *Istr* and returns it into *right*.

Remarks

The valid input formats are

- *(real part, imaginary part)*
- *(real part)*
- *real part*

Example

```
C++
```

```
// complex_op_extract.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    double pi = 3.14159265359;

    complex <double> c2;

    cout << "Input a complex number ( try: 2.0 ): ";
    cin >> c2;
    cout << c2 << endl;
}
```

Output

```
Input a complex number ( try: 2.0 ): 2.0
2.0
```

`complex` Class

Article • 03/09/2022

The class template describes an object that stores two objects of type `Type`, one that represents the real part of a complex number and one that represents the imaginary part.

Syntax

C++

```
template <class Type>
class complex
```

Remarks

An object of class `Type`:

- Has a public default constructor, destructor, copy constructor, and assignment operator with conventional behavior.
- Can be assigned integer or floating-point values, or type cast to such values with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class `Type` may throw exceptions.

Explicit specializations of class template `complex` exist for the three floating-point types. In this implementation, a value of any other type `Type` is typecast to **`double`** for actual calculations, with the **`double`** result assigned back to the stored object of type `Type`.

Members

Constructors

Name	Description
complex	Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

Typedefs

Name	Description
value_type	A type that represents the data type used to represent the real and imaginary parts of a complex number.

Functions

Name	Description
imag	Extracts the imaginary component of a complex number.
real	Extracts the real component of a complex number.

Operators

Name	Description
operator*=	Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
operator+=	Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it's added.
operator-=	Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it's added.
operator/=	Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.
operator=	Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it's being assigned.

`complex`

Constructs a complex number with specified real and imaginary parts or as a copy of some other complex number.

```
C++
```

```
constexpr complex(
    const T& _RealVal = 0,
    const T& _ImagVal = 0);

template <class Other>
constexpr complex(
    const complex<Other>& complexNum);
```

Parameters

`_RealVal`

The value of the real part used to initialize the complex number being constructed.

`_ImagVal`

The value of the imaginary part used to initialize the complex number being constructed.

`complexNum`

The complex number whose real and imaginary parts are used to initialize the complex number being constructed.

Remarks

The first constructor initializes the stored real part to *`_RealVal`* and the stored imaginary part to *`_Imagval`*. The second constructor initializes the stored real part to `complexNum.real()` and the stored imaginary part to `complexNum.imag()`.

In this implementation, if a translator doesn't support member template functions, the template:

```
C++
```

```
template <class Other>
complex(const complex<Other>& right);
```

is replaced with:

```
C++
```

```
complex(const complex& right);
```

which is the copy constructor.

Example

C++

```
// complex_complex.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex<double> c1( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,"
        << "c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of another complex number
    complex<double> c2( c1 );
    cout << "Initializing with the real and imaginary parts of c1,"
        << " c2 = " << c2 << endl;

    // Complex numbers can be initialized in polar form
    // but will be stored in Cartesian form
    complex<double> c3( polar( sqrt( (double)8 ) , pi / 4 ) );
    cout << "c3 = polar( sqrt( 8 ) , pi / 4 ) = " << c3 << endl;

    // The modulus and argument of a complex number can be recovered
    double absc3 = abs( c3 );
    double argc3 = arg( c3 );
    cout << "The modulus of c3 is recovered from c3 using: abs( c3 ) = "
        << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg( c3 ) = "
        << argc3 << " radians, which is " << argc3 * 180 / pi
        << " degrees." << endl;
}
```

`imag`

Extracts the imaginary component of a complex number.

```
```

C++

```
T imag() const;

T imag(const T& right);
```

Parameters

right

A complex number whose imaginary value is to be extracted.

Return Value

The imaginary part of the complex number.

Remarks

For a complex number *a* + *bi*, the imaginary part or component is *Im(a + bi) = b*.

Example

C++

```
// complex_imag.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;

    complex<double> c1( 4.0 , 3.0 );
    cout << "The complex number c1 = " << c1 << endl;

    double dr1 = c1.real();
    cout << "The real part of c1 is c1.real() = "
         << dr1 << "." << endl;

    double di1 = c1.imag();
    cout << "The imaginary part of c1 is c1.imag() = "
         << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is c1.real() = 4.
The imaginary part of c1 is c1.imag() = 3.
```

`operator*=`

Multiplies a target complex number by a factor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

C++

```
template <class Other>
complex& operator*=(const complex<Other>& right);

complex<Type>& operator*=(const Type& right);

complex<Type>& operator*=(const complex<Type>& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been multiplied by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_me.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main()
```

```
{
    using namespace std;
    double pi = 3.14159265359;

    // Example of the first member function
    // type complex<double> multiplied by type complex<double>
    complex<double> cl1( polar ( 3.0 , pi / 6 ) );
    complex<double> cr1( polar ( 2.0 , pi / 3 ) );
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = cl1 * cr1;
    cout << "Quotient of two complex numbers is: cs1 = cl1 * cr1 = "
        << cs1 << endl;

    // This is equivalent to the following operation
    cl1 *= cr1;
    cout << "Quotient of two complex numbers is also: cl1 *= cr1 = "
        << cl1 << endl;

    double abscl1 = abs ( cl1 );
    double argcl1 = arg ( cl1 );
    cout << "The modulus of cl1 is: " << abscl1 << endl;
    cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type complex<double> multiplied by type double
    complex<double> cl2 ( polar ( 3.0 , pi / 6 ) );
    double cr2 = 5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = cl2 * cr2;
    cout << "Quotient of two complex numbers is: cs2 = cl2 * cr2 = "
        << cs2 << endl;

    // This is equivalent to the following operation
    cl2 *= cr2;
    cout << "Quotient of two complex numbers is also: cl2 *= cr2 = "
        << cl2 << endl;

    double abscl2 = abs ( cl2 );
    double argcl2 = arg ( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
        << argcl2 * 180 / pi << " degrees." << endl;
}
```

operator+=

Adds a number to a target complex number, where the number added may be complex or of the same type as are the real and imaginary parts of the complex number to which it's added.

```
C++
```

```
template <class Other>
complex<Type>& operator+=(const complex<Other>& right);

complex<Type>& operator+=(const Type& right);

complex<Type>& operator+=(const complex<Type>& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has had the number specified as a parameter added.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

```
C++
```

```
// complex_op_pe.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> added to type complex<double>
   complex<double> c1( 3.0 , 4.0 );
```

```
    complex<double> cr1( 2.0 , -1.0 );
    cout << "The left-side complex number is cl1 = " << cl1 << endl;
    cout << "The right-side complex number is cr1 = " << cr1 << endl;

    complex<double> cs1 = cl1 + cr1;
    cout << "The sum of the two complex numbers is: cs1 = cl1 + cr1 = "
         << cs1 << endl;

    // This is equivalent to the following operation
    cl1 += cr1;
    cout << "The complex number cr1 added to the complex number cl1 is:"
         << "\n cl1 += cr1 = " << cl1 << endl;

    double abscl1 = abs( cl1 );
    double argcl1 = arg( cl1 );
    cout << "The modulus of cl1 is: " << abscl1 << endl;
    cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
         << argcl1 * 180 / pi << " degrees." << endl << endl;

    // Example of the second member function
    // type double added to type complex<double>
    complex<double> cl2( -2 , 4 );
    double cr2 =5.0;
    cout << "The left-side complex number is cl2 = " << cl2 << endl;
    cout << "The right-side complex number is cr2 = " << cr2 << endl;

    complex<double> cs2 = cl2 + cr2;
    cout << "The sum of the two complex numbers is: cs2 = cl2 + cr2 = "
         << cs2 << endl;

    // This is equivalent to the following operation
    cl2 += cr2;
    cout << "The complex number cr2 added to the complex number cl2 is:"
         << "\n cl2 += cr2 = " << cl2 << endl;

    double abscl2 = abs( cl2 );
    double argcl2 = arg( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The sum of the two complex numbers is: cs1 = cl1 + cr1 = (5,3)
The complex number cr1 added to the complex number cl1 is:
cl1 += cr1 = (5,3)
The modulus of cl1 is: 5.83095
The argument of cl1 is: 0.54042 radians, which is 30.9638 degrees.

The left-side complex number is cl2 = (-2,4)
```

```
The right-side complex number is cr2 = 5
The sum of the two complex numbers is: cs2 = cl2 + cr2 = (3,4)
The complex number cr2 added to the complex number cl2 is:
cl2 += cr2 = (3,4)
The modulus of cl2 is: 5
The argument of cl2 is: 0.927295 radians, which is 53.1301 degrees.
```

`operator-=`

Subtracts a number from a target complex number, where the number subtracted may be complex or of the same type as are the real and imaginary parts of the complex number to which it's added.

C++

```
template <class Other>
complex<Type>& operator-=(const complex<Other>& complexNum);

complex<Type>& operator-=(const Type& _RealPart);

complex<Type>& operator-=(const complex<Type>& complexNum);
```

Parameters

`complexNum`

A complex number to be subtracted from the target complex number.

`_RealPart`

A real number to be subtracted from the target complex number.

Return Value

A complex number that has had the number specified as a parameter subtracted from it.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```cpp
// complex_op_se.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> subtracted from type complex<double>
   complex<double> cl1( 3.0 , 4.0 );
   complex<double> cr1( 2.0 , -1.0 );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex<double> cs1 = cl1 - cr1;
   cout << "The difference between the two complex numbers is:"
        << "\n cs1 = cl1 - cr1 = " << cs1 << endl;

   // This is equivalent to the following operation
   cl1 -= cr1;
   cout << "Complex number cr1 subtracted from complex number cl1 is:"
        << "\n cl1 -= cr1 = " << cl1 << endl;

   double abscl1 = abs( cl1 );
   double argcl1 = arg( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type double subtracted from type complex<double>
   complex<double> cl2( 2.0 , 4.0 );
   double cr2 = 5.0;
   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;

   complex<double> cs2 = cl2 - cr2;
   cout << "The difference between the two complex numbers is:"
        << "\n cs2 = cl2 - cr2 = " << cs2 << endl;

   // This is equivalent to the following operation
   cl2  -= cr2;
   cout << "Complex number cr2 subtracted from complex number cl2 is:"
        << "\n cl2 -= cr2 = " << cl2 << endl;

   double abscl2 = abs( cl2 );
   double argcl2 = arg( cl2 );
   cout << "The modulus of cl2 is: " << abscl2 << endl;
   cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
        << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The difference between the two complex numbers is:
cs1 = cl1 - cr1 = (1,5)
Complex number cr1 subtracted from complex number cl1 is:
cl1 -= cr1 = (1,5)
The modulus of cl1 is: 5.09902
The argument of cl1 is: 1.3734 radians, which is 78.6901 degrees.

The left-side complex number is cl2 = (2,4)
The right-side complex number is cr2 = 5
The difference between the two complex numbers is:
cs2 = cl2 - cr2 = (-3,4)
Complex number cr2 subtracted from complex number cl2 is:
cl2 -= cr2 = (-3,4)
The modulus of cl2 is: 5
The argument of cl2 is: 2.2143 radians, which is 126.87 degrees.
```

`operator/=`

Divides a target complex number by a divisor, which may be complex or be the same type as are the real and imaginary parts of the complex number.

C++

```
template <class Other>
complex<Type>& operator/=(const complex<Other>& complexNum);

complex<Type>& operator/=(const Type& _RealPart);

complex<Type>& operator/=(const complex<Type>& complexNum);
```

Parameters

`complexNum`

A complex number to be subtracted from the target complex number.

`_RealPart`

A real number to be subtracted from the target complex number.

Return Value

A complex number that has been divided by the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_de.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> divided by type complex<double>
   complex<double> cl1( polar (3.0 , pi / 6 ) );
   complex<double> cr1( polar (2.0 , pi / 3 ) );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   complex<double> cs1 = cl1 / cr1;
   cout << "The quotient of the two complex numbers is: cs1 = cl1 /cr1 = "
        << cs1 << endl;

   // This is equivalent to the following operation
   cl1 /= cr1;
   cout << "Quotient of two complex numbers is also: cl1 /= cr1 = "
        << cl1 << endl;

   double abscl1 = abs( cl1 );
   double argcl1 = arg( cl1 );
   cout << "The modulus of cl1 is: " << abscl1 << endl;
   cout << "The argument of cl1 is: "<< argcl1 << " radians, which is "
        << argcl1 * 180 / pi << " degrees." << endl << endl;

   // Example of the second member function
   // type complex<double> divided by type double
   complex<double> cl2( polar(3.0 , pi / 6 ) );
   double cr2 =5;
   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;

   complex<double> cs2 = cl2 / cr2;
   cout << "The quotient of the two complex numbers is: cs2 /= cl2 cr2 = "
        << cs2 << endl;
```

```
    // This is equivalent to the following operation
    cl2 /= cr2;
    cout << "Quotient of two complex numbers is also: cl2 = /cr2 = "
         << cl2 << endl;

    double abscl2 = abs( cl2 );
    double argcl2 = arg( cl2 );
    cout << "The modulus of cl2 is: " << abscl2 << endl;
    cout << "The argument of cl2 is: "<< argcl2 << " radians, which is "
         << argcl2 * 180 / pi << " degrees." << endl << endl;
}
```

Output

```
The left-side complex number is cl1 = (2.59808,1.5)
The right-side complex number is cr1 = (1,1.73205)
The quotient of the two complex numbers is: cs1 = cl1 /cr1 = (1.29904,-0.75)
Quotient of two complex numbers is also: cl1 /= cr1 = (1.29904,-0.75)
The modulus of cl1 is: 1.5
The argument of cl1 is: -0.523599 radians, which is -30 degrees.

The left-side complex number is cl2 = (2.59808,1.5)
The right-side complex number is cr2 = 5
The quotient of the two complex numbers is: cs2 /= cl2 cr2 = (0.519615,0.3)
Quotient of two complex numbers is also: cl2 = /cr2 = (0.519615,0.3)
The modulus of cl2 is: 0.6
The argument of cl2 is: 0.523599 radians, which is 30 degrees.
```

`operator=`

Assigns a number to a target complex number, where the number assigned may be complex or of the same type as are the real and imaginary parts of the complex number to which it's being assigned.

C++

```
template <class Other>
complex<Type>& operator=(const complex<Other>& right);

complex<Type>& operator=(const Type& right);
```

Parameters

right

A complex number or a number that is of the same type as the parameter of the target complex number.

Return Value

A complex number that has been assigned the number specified as a parameter.

Remarks

The operation is overloaded so that simple arithmetic operations can be executed without the conversion of the data to a particular format.

Example

C++

```
// complex_op_as.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // Example of the first member function
   // type complex<double> assigned to type complex<double>
   complex<double> cl1( 3.0 , 4.0 );
   complex<double> cr1( 2.0 , -1.0 );
   cout << "The left-side complex number is cl1 = " << cl1 << endl;
   cout << "The right-side complex number is cr1 = " << cr1 << endl;

   cl1  = cr1;
   cout << "The complex number cr1 assigned to the complex number cl1 is:"
        << "\ncl1 = cr1 = " << cl1 << endl;

   // Example of the second member function
   // type double assigned to type complex<double>
   complex<double> cl2( -2 , 4 );
   double cr2 =5.0;
   cout << "The left-side complex number is cl2 = " << cl2 << endl;
   cout << "The right-side complex number is cr2 = " << cr2 << endl;

   cl2 = cr2;
   cout << "The complex number cr2 assigned to the complex number cl2 is:"
        << "\ncl2 = cr2 = " << cl2 << endl;

   cl2 = complex<double>(3.0, 4.0);
   cout << "The complex number (3, 4) assigned to the complex number cl2
is:"
        << "\ncl2 = " << cl2 << endl;
}
```

Output

```
The left-side complex number is cl1 = (3,4)
The right-side complex number is cr1 = (2,-1)
The complex number cr1 assigned to the complex number cl1 is:
cl1 = cr1 = (2,-1)
The left-side complex number is cl2 = (-2,4)
The right-side complex number is cr2 = 5
The complex number cr2 assigned to the complex number cl2 is:
cl2 = cr2 = (5,0)
The complex number (3, 4) assigned to the complex number cl2 is:
cl2 = (3,4)
```

`real`

Gets or sets the `real` component of a complex number.

C++

```
constexpr T real() const;

T real(const T& right);
```

Parameters

`right`

A complex number whose `real` value is to be extracted.

Return Value

The `real` part of the complex number.

Remarks

For a complex number *a* + *bi*, the `real` part or component is *Re(a + bi) = a*.

Example

C++

```
// complex_class_real.cpp
// compile with: /EHsc
```

```
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;

   complex<double> c1( 4.0 , 3.0 );
   cout << "The complex number c1 = " << c1 << endl;

   double dr1 = c1.real();
   cout << "The real part of c1 is c1.real() = "
        << dr1 << "." << endl;

   double di1 = c1.imag();
   cout << "The imaginary part of c1 is c1.imag() = "
        << di1 << "." << endl;
}
```

Output

```
The complex number c1 = (4,3)
The real part of c1 is c1.real() = 4.
The imaginary part of c1 is c1.imag() = 3.
```

`value_type`

A type that represents the data type used to represent the real and imaginary parts of a complex number.

C++

```
typedef Type value_type;
```

Remarks

`value_type` is a synonym for the class complex `Type` template parameter.

Example

C++

```
// complex_valuetype.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>
```

```
int main( )
{
    using namespace std;
    complex<double>::value_type a = 3, b = 4;

    complex<double> c1 ( a , b );
    cout << "Specifying initial real & imaginary parts"
        << "\nof type value_type: "
        << "c1 = " << c1 << "." << endl;
}
```

Output

```
Specifying initial real & imaginary parts
of type value_type: c1 = (3,4).
```

See also

Thread Safety in the C++ Standard Library

`complex<double>`

Article • 12/06/2021

Describes an object that stores an ordered pair of objects both of type `double`, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

C++

```
template <>
class complex<double> {
public:
    constexpr complex(
    double RealVal = 0,
    double ImagVal = 0);

constexpr complex(const complex<double>& complexNum);

constexpr explicit complex(const complex<long double>& complexNum);
// rest same as class template complex
};
```

Parameters

RealVal
The value of type `double` for the real part of the complex number being constructed.

ImagVal
The value of type `double` for the imaginary part of the complex number being constructed.

complexNum
The complex number of type `float` or of type `long double` whose real and imaginary parts are used to initialize a complex number of type `double` being constructed.

Return Value

A complex number of type `double`.

Remarks

The explicit specialization of the class template complex to a complex class of type `double` differs from the class template only in the constructors it defines. The conversion from `float` to `double` is allowed to be implicit, but the conversion from `long double` to `double` is required to be `explicit`. The use of `explicit` rules out the initiation with type conversion using assignment syntax.

For more information on the class template `complex`, see complex Class. For a list of members of the class template `complex`, see .

Example

C++

```
// complex_comp_dbl.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <double> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,\n"
        << "as type double gives c1 = " << c1 << endl;

   // The second constructor initializes values of the real &
   // imaginary parts using those of complex number of type float
   complex <float> c2float ( 4.0 , 5.0 );
   complex <double> c2double ( c2float );
   cout << "Implicit conversion from type float to type double,"
        << endl << "gives c2double = " << c2double << endl;

   // The third constructor initializes values of the real &
   // imaginary parts using those of a complex number
   // of type long double
   complex <long double> c3longdouble ( 4.0 , 5.0 );
   complex <double> c3double ( c3longdouble );
   cout << "Explicit conversion from type float to type double,"
        << endl << "gives c3longdouble = " << c3longdouble << endl;

   // The modulus and argument of a complex number can be recovered
   double absc3 = abs ( c3longdouble );
   double argc3 = arg ( c3longdouble );
   cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
```

```
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:" << endl
         << "arg ( c3 ) = " << argc3 << " radians, which is "
         << argc3 * 180 / pi << " degrees." << endl;
}
/* Output:
Specifying initial real & imaginary parts,
as type double gives c1 = (4,5)
Implicit conversion from type float to type double,
gives c2double = (4,5)
Explicit conversion from type float to type double,
gives c3longdouble = (4,5)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 6.40312
Argument of c3 is recovered from c3 using:
arg ( c3 ) = 0.896055 radians, which is 51.3402 degrees.
*/
```

Requirements

Header: <complex>

Namespace: std

See also

complex Class

Thread Safety in the C++ Standard Library

`complex<float>`

Article • 12/06/2021

Describes an object that stores an ordered pair of objects both of type `float`, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

C++

```
template <>
class complex<float> {
public:
    constexpr complex(
    float _RealVal = 0,
    float _ImagVal = 0);

constexpr complex(
    const complex<float>& complexNum);

constexpr complex(
    const complex<double>& complexNum);

constexpr complex(
    const complex<long double>& complexNum);
// rest same as class template complex
};
```

Parameters

_RealVal
The value of type `float` for the real part of the complex number being constructed.

_ImagVal
The value of type `float` for the imaginary part of the complex number being constructed.

complexNum
The complex number of type `double` or of type `long double` whose real and imaginary parts are used to initialize a complex number of type `float` being constructed.

Return Value

A complex number of type `float`.

Remarks

The explicit specialization of the class template complex to a complex class of type `float` differs from the class template only in the constructors it defines. The conversion from `float` to `double` is allowed to be implicit, but the less safe conversion from `float` to `long double` is required to be `explicit`. The use of `explicit` rules out the initiation with type conversion using assignment syntax.

For more information on the class template `complex`, see complex Class. For a list of members of the class template `complex`, see .

Example

C++

```
// complex_comp_flt.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;

   // The first constructor specifies real & imaginary parts
   complex <float> c1 ( 4.0 , 5.0 );
   cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;

   // The second constructor initializes values of the real &
   // imaginary parts using those of complex number of type double
   complex <double> c2double ( 1.0 , 3.0 );
   complex <float> c2float ( c2double );
   cout << "Implicit conversion from type double to type float,"
        << endl << "gives c2float = " << c2float << endl;

   // The third constructor initializes values of the real &
   // imaginary parts using those of a complex number
   // of type long double
   complex <long double> c3longdouble ( 3.0 , 4.0 );
   complex <float> c3float ( c3longdouble );
   cout << "Explicit conversion from type long double to type float,"
```

```
        << endl << "gives c3float = " << c3float << endl;

   // The modulus and argument of a complex number can be recovered
   double absc3 = abs ( c3float);
   double argc3 = arg ( c3float);
   cout << "The modulus of c3 is recovered from c3 using: abs ( c3 ) = "
        << absc3 << endl;
   cout << "Argument of c3 is recovered from c3 using:"
        << endl << "arg ( c3 ) = "
        << argc3 << " radians, which is " << argc3 * 180 / pi
        << " degrees." << endl;
}
/* Output:
Specifying initial real & imaginary parts,
as type float gives c1 = (4,5)
Implicit conversion from type double to type float,
gives c2float = (1,3)
Explicit conversion from type long double to type float,
gives c3float = (3,4)
The modulus of c3 is recovered from c3 using: abs ( c3 ) = 5
Argument of c3 is recovered from c3 using:
arg ( c3 ) = 0.927295 radians, which is 53.1301 degrees.
*/
```

Requirements

Header: <complex>

Namespace: std

See also

complex Class

Thread Safety in the C++ Standard Library

`complex<long double>`

Article • 12/06/2021

This explicitly specialized class template describes an object that stores an ordered pair of objects, both of type `long double`, the first representing the real part of a complex number and the second representing the imaginary part.

Syntax

C++

```
template <>
class complex<long double> {
public:
    constexpr complex(
    long double _RealVal = 0,
    long double _ImagVal = 0);

complex(
    constexpr complex<long double>& complexNum);

// rest same as class template complex
};
```

Parameters

_RealVal
The value of type `long double` for the real part of the complex number being constructed.

_ImagVal
The value of type `long double` for the imaginary part of the complex number being constructed.

complexNum
The complex number of type `double` or of type `float` whose real and imaginary parts are used to initialize a complex number of type `long double` being constructed.

Return Value

A complex number of type `long double`.

Remarks

The explicit specialization of the class template `complex` to a complex class of type `long double` differs from the class template only in the constructors it defines. The conversion from `long double` to `float` is allowed to be implicit, but the conversion from `double` to `long double` is required to be `explicit`. The use of `explicit` rules out the initiation with type conversion using assignment syntax.

For more information on the class template `complex` and its members, see complex Class.

Microsoft-specific: The `long double` and `double` types have the same representation, but are distinct types. For more information, see Built-in types.

Example

C++

```
// complex_comp_ld.cpp
// compile with: /EHsc
#include <complex>
#include <iostream>

int main( )
{
    using namespace std;
    double pi = 3.14159265359;

    // The first constructor specifies real & imaginary parts
    complex<long double> c1( 4.0 , 5.0 );
    cout << "Specifying initial real & imaginary parts,\n"
        << " as type float gives c1 = " << c1 << endl;

    // The second constructor initializes values of the real &
    // imaginary parts using those of complex number of type float
    complex<float> c2float( 1.0 , 3.0 );
    complex<long double> c2longdouble ( c2float );
    cout << "Implicit conversion from type float to type long double,"
        << "\n gives c2longdouble = " << c2longdouble << endl;

    // The third constructor initializes values of the real &
    // imaginary parts using those of a complex number
    // of type double
    complex<double> c3double( 3.0 , 4.0 );
    complex<long double> c3longdouble( c3double );
    cout << "Implicit conversion from type long double to type float,"
        << "\n gives c3longdouble = " << c3longdouble << endl;
```

```
    // The modulus and argument of a complex number can be recovered
    double absc3 = abs( c3longdouble );
    double argc3 = arg( c3longdouble );
    cout << "The modulus of c3 is recovered from c3 using: abs( c3 ) = "
         << absc3 << endl;
    cout << "Argument of c3 is recovered from c3 using:\n arg( c3 ) = "
         << argc3 << " radians, which is " << argc3 * 180 / pi
         << " degrees." << endl;
}
```

Output

```
Specifying initial real & imaginary parts,
as type float gives c1 = (4,5)
Implicit conversion from type float to type long double,
gives c2longdouble = (1,3)
Implicit conversion from type long double to type float,
gives c3longdouble = (3,4)
The modulus of c3 is recovered from c3 using: abs( c3 ) = 5
Argument of c3 is recovered from c3 using:
arg( c3 ) = 0.927295 radians, which is 53.1301 degrees.
```

Requirements

Header: <complex>

Namespace: std

See also

complex Class

Thread Safety in the C++ Standard Library

<condition_variable>

Article • 12/07/2021

Defines the classes condition_variable and condition_variable_any that are used to create objects that wait for a condition to become true.

This header uses Concurrency Runtime (ConcRT) so that you can use it together with other ConcRT mechanisms. For more information about ConcRT, see Concurrency Runtime.

Requirements

Header: <condition_variable>

Namespace: std

> ⓘ Note
>
> In code that is compiled by using **/clr**, this header is blocked.

Remarks

Code that waits for a condition variable must also use a `mutex`. A calling thread must lock the `mutex` before it calls the functions that wait for the condition variable. The `mutex` is then locked when the called function returns. The `mutex` is not locked while the thread waits for the condition to become true. So that there are no unpredictable results, each thread that waits for a condition variable must use the same `mutex` object.

Objects of type `condition_variable_any` can be used with a mutex of any type. The type of the mutex that is used does not have to provide the `try_lock` method. Objects of type `condition_variable` can only be used with a mutex of type `unique_lock<mutex>`. Objects of this type may be faster than objects of type `condition_variable_any<unique_lock<mutex>>`.

To wait for an event, first lock the mutex, and then call one of the `wait` methods on the condition variable. The `wait` call blocks until another thread signals the condition variable.

Spurious wakeups occur when threads that are waiting for condition variables become unblocked without appropriate notifications. To recognize such spurious wakeups, code

that waits for a condition to become true should explicitly check that condition when the code returns from a wait function. This is usually done by using a loop; you can use `wait(unique_lock<mutex>& lock, Predicate pred)` to perform this loop for you.

```
C++
```

```
while (condition is false)
    wait for condition variable;
```

The `condition_variable_any` and `condition_variable` classes each have three methods that wait for a condition.

- `wait` waits for an unbounded time period.
- `wait_until` waits until a specified `time`.
- `wait_for` waits for a specified `time interval`.

Each of these methods has two overloaded versions. One just waits and can wake up spuriously. The other takes an additional template argument that defines a predicate. The method does not return until the predicate is **`true`**.

Each class also has two methods that are used to notify a condition variable that its condition is **`true`**.

- `notify_one` wakes up one of the threads that is waiting for the condition variable.
- `notify_all` wakes up all of the threads that are waiting for the condition variable.

Functions and Enums

```
C++
```

```
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex>
lk);

enum class cv_status { no_timeout, timeout };
```

See also

Header Files Reference
condition_variable Class
condition_variable_any Class

`<condition_variable>` enums

Article • 12/07/2021

cv_status

Supplies symbolic names for the return values of the methods of class template condition_variable.

class cv_status { no_timeout timeout };

condition_variable Class

Article • 08/03/2021

Use the `condition_variable` class to wait for an event when you have a `mutex` of type `unique_lock<mutex>`. Objects of this type may have better performance than objects of type condition_variable_any<unique_lock<mutex>>.

Syntax

```
C++
class condition_variable;
```

Members

Constructors

Name	Description
condition_variable	Constructs a `condition_variable` object.

Functions

Name	Description
native_handle	Returns the implementation-specific type representing the condition_variable handle.
notify_all	Unblocks all threads that are waiting for the `condition_variable` object.
notify_one	Unblocks one of the threads that are waiting for the `condition_variable` object.
wait	Blocks a thread.
wait_for	Blocks a thread, and sets a time interval after which the thread unblocks.
wait_until	Blocks a thread, and sets a maximum point in time at which the thread unblocks.

condition_variable

Constructs a `condition_variable` object.

```
C++

condition_variable();
```

Remarks

If not enough memory is available, the constructor throws a system_error object that has a `not_enough_memory` error code. If the object cannot be constructed because some other resource is not available, the constructor throws a `system_error` object that has a `resource_unavailable_try_again` error code.

native_handle

Returns the implementation-specific type that represents the condition_variable handle.

```
C++

native_handle_type native_handle();
```

Return Value

`native_handle_type` is defined as a pointer to Concurrency Runtime internal data structures.

notify_all

Unblocks all threads that are waiting for the `condition_variable` object.

```
C++

void notify_all() noexcept;
```

notify_one

Unblocks one of the threads that are waiting on the `condition_variable` object.

```
C++
```

```
void notify_one() noexcept;
```

wait

Blocks a thread.

```
C++
```

```
void wait(unique_lock<mutex>& Lck);

template <class Predicate>
void wait(unique_lock<mutex>& Lck, Predicate Pred);
```

Parameters

Lck
A unique_lock<mutex> object.

Pred
Any expression that returns `true` or `false`.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to notify_one or notify_all. It can also wake up spuriously.

In effect, the second method executes the following code.

```
C++
```

```
while(!Pred())
    wait(Lck);
```

wait_for

Blocks a thread, and sets a time interval after which the thread unblocks.

```
C++
```

```
template <class Rep, class Period>
cv_status wait_for(
    unique_lock<mutex>& Lck,
```

```
    const chrono::duration<Rep, Period>& Rel_time);

template <class Rep, class Period, class Predicate>
bool wait_for(
    unique_lock<mutex>& Lck,
    const chrono::duration<Rep, Period>& Rel_time,
    Predicate Pred);
```

Parameters

Lck
A unique_lock<mutex> object.

Rel_time
A `chrono::duration` object that specifies the amount of time before the thread wakes up.

Pred
Any expression that returns **`true`** or **`false`**.

Return Value

The first method returns `cv_status::timeout` if the wait terminates when *Rel_time* has elapsed. Otherwise, the method returns `cv_status::no_timeout`.

The second method returns the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to notify_one or notify_all or until the time interval *Rel_time* has elapsed. It can also wake up spuriously.

In effect, the second method executes the following code.

C++

```
while(!Pred())
    if(wait_for(Lck, Rel_time) == cv_status::timeout)
    return Pred();

return true;
```

wait_until

Blocks a thread, and sets a maximum point in time at which the thread unblocks.

C++

```
template <class Clock, class Duration>
cv_status wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time);

template <class Clock, class Duration, class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time,
    Predicate Pred);

cv_status wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time);

template <class Predicate>
bool wait_until(
    unique_lock<mutex>& Lck,
    const xtime* Abs_time,
    Predicate Pred);
```

Parameters

Lck
A unique_lock<mutex> object.

Abs_time
A chrono::time_point object.

Pred
Any expression that returns `true` or `false`.

Return Value

Methods that return a `cv_status` type return `cv_status::timeout` if the wait terminates when *Abs_time* elapses. Otherwise, the methods return `cv_status::no_timeout`.

Methods that return a `bool` return the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to notify_one or notify_all or until `Abs_time`. It can also wake up spuriously.

In effect, the second method executes the following code

```
C++
```

```
while(!Pred())
    if(wait_until(Lck, Abs_time) == cv_status::timeout)
    return Pred();

return true;
```

The third and fourth methods use a pointer to an object of type `xtime` to replace the `chrono::time_point` object. The `xtime` object specifies the maximum amount of time to wait for a signal.

See also

Header Files Reference
<condition_variable>

condition_variable_any Class

Article • 08/03/2021

Use the class `condition_variable_any` to wait for an event that has any `mutex` type.

Syntax

```
C++
```

```
class condition_variable_any;
```

Members

Constructors

Name	Description
condition_variable_any	Constructs a `condition_variable_any` object.

Functions

Name	Description
notify_all	Unblocks all threads that are waiting for the `condition_variable_any` object.
notify_one	Unblocks one of the threads that are waiting for the `condition_variable_any` object.
wait	Blocks a thread.
wait_for	Blocks a thread, and sets a time interval after which the thread unblocks.
wait_until	Blocks a thread, and sets a maximum point in time at which the thread unblocks.

condition_variable_any

Constructs a `condition_variable_any` object.

```
C++
```

```
condition_variable_any();
```

Remarks

If not enough memory is available, the constructor throws a system_error object that has a `not_enough_memory` error code. If the object cannot be constructed because some other resource is not available, the constructor throws a `system_error` object that has a `resource_unavailable_try_again` error code.

notify_all

Unblocks all threads that are waiting for the `condition_variable_any` object.

C++

```
void notify_all() noexcept;
```

notify_one

Unblocks one of the threads that are waiting on the `condition_variable_any` object.

C++

```
void notify_one() noexcept;
```

wait

Blocks a thread.

C++

```
template <class Lock>
void wait(Lock& Lck);

template <class Lock, class Predicate>
void wait(Lock& Lck, Predicate Pred);
```

Parameters

Lck
A `mutex` object of any type.

Pred
Any expression that returns `true` or `false`.

Remarks

The first method blocks until the `condition_variable_any` object is signaled by a call to notify_one or notify_all. It can also wake up spuriously.

The second method in effect executes the following code.

```
C++
```

```
while (!Pred())
    wait(Lck);
```

wait_for

Blocks a thread, and sets a time interval after which the thread unblocks.

```
C++
```

```
template <class Lock, class Rep, class Period>
bool wait_for(Lock& Lck, const chrono::duration<Rep, Period>& Rel_time);

template <class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& Lck, const chrono::duration<Rep, Period>& Rel_time,
Predicate Pred);
```

Parameters

Lck
A `mutex` object of any type.

Rel_time
A `chrono::duration` object that specifies the amount of time before the thread wakes up.

Pred
Any expression that returns `true` or `false`.

Return Value

The first method returns `cv_status::timeout` if the wait terminates when *Rel_time* has elapsed. Otherwise, the method returns `cv_status::no_timeout`.

The second method returns the value of *Pred*.

Remarks

The first method blocks until the `condition_variable_any` object is signaled by a call to notify_one or notify_all, or until the time interval *Rel_time* has elapsed. It can also wake up spuriously.

The second method in effect executes the following code.

C++

```
while(!Pred())
    if(wait_for(Lck, Rel_time) == cv_status::timeout)
    return Pred();

return true;
```

wait_until

Blocks a thread, and sets a maximum point in time at which the thread unblocks.

C++

```
template <class Lock, class Clock, class Duration>
void wait_until(Lock& Lck, const chrono::time_point<Clock, Duration>&
Abs_time);

template <class Lock, class Clock, class Duration, class Predicate>
void wait_until(
    Lock& Lck,
    const chrono::time_point<Clock, Duration>& Abs_time,
    Predicate Pred);

template <class Lock>
void wait_until(Lock Lck, const xtime* Abs_time);

template <class Lock, class Predicate>
void wait_until(
    Lock Lck,
    const xtime* Abs_time,
    Predicate Pred);
```

Parameters

Lck
A mutex object.

Abs_time
A chrono::time_point object.

Pred
Any expression that returns `true` or `false`.

Return Value

Methods that return a `cv_status` type return `cv_status::timeout` if the wait terminates when *Abs_time* elapses. Otherwise, the methods return `cv_status::no_timeout`.

Methods that return a `bool` return the value of *Pred*.

Remarks

The first method blocks until the `condition_variable` object is signaled by a call to notify_one or notify_all, or until *Abs_time*. It can also wake up spuriously.

The second method in effect executes the following code.

```
C++

while(!Pred())
    if(wait_until(Lck, Abs_time) == cv_status::timeout)
    return Pred();

return true;
```

The third and fourth methods use a pointer to an object of type `xtime` to replace the `chrono::time_point` object. The `xtime` object specifies the maximum amount of time to wait for a signal.

<csetjmp>

Article • 12/07/2021

Includes the Standard C library header <setjmp.h> and adds the associated names to the `std` namespace.

Syntax

```
C++

#include <csetjmp>

using jmp_buf = see below;
```

Functions

```
C++

[[noreturn]] void longjmp(jmp_buf env, int val);
```

Macros

```
C++

#define setjmp(env)
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<csignal>

Article • 12/07/2021

Includes the C Standard library header <signal.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Syntax

C++

```
#include <csignal>
```

Namespace and Macros

C++

```
namespace std {
    using sig_atomic_t = see below;

    extern using signal-handler = void(int);
}

#define SIG_DFL
#define SIG_ERR
#define SIG_IGN
#define SIGABRT
#define SIGFPE
#define SIGILL
#define SIGINT
#define SIGSEGV
#define SIGTERM
```

Functions

C++

```
signal-handler* signal(int sig, signal-handler* func);
int raise(int sig);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cstdalign>

Article • 08/03/2021

In some C++ standard library implementations, this header includes the C standard library header <stdalign.h>, and adds the associated names to the std namespace. Because that header isn't implemented in MSVC, the <cstdalign> header defines compatibility macros __alignas_is_defined and __alignof_is_defined.

> **Note**
>
> Because the <stdalign.h> header defines macros that are keywords in C++, including it has no effect. The <stdalign.h> header is deprecated in C++. The <cstdalign> header is deprecated in C++17 and removed in the draft C++20 standard.

Requirements

Header: <cstdalign>

Namespace: std

Macros

Macro	Description
__alignas_is_defined	A C compatibility macro that expands to the integer constant 1.
__alignof_is_defined	A C compatibility macro that expands to the integer constant 1.

See also

Header files reference
C++ standard library overview
Thread safety in the C++ standard library

<cstdarg>

Article • 12/07/2021

Includes the C Standard library header <stdarg.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C Standard library header are declared in the `std` namespace.

Syntax

```
C++

#include <cstdarg>
```

Namespace and Macros

```
C++

namespace std {
    using va_list = see below;
}

#define va_arg(V, P)
#define va_copy(VDST, VSRC)
#define va_end(V)
#define va_start(V, P)
```

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

`<cstdbool>`

Article • 12/07/2021

Includes the C standard library header <stdbool.h> and adds the associated names to the `std` namespace.

> **Note**
>
> Because the <stdbool.h> header defines macros that are keywords in C++, including it has no effect. The <stdbool.h> header is deprecated in C++. The <cstdbool> header is deprecated in C++17 and removed in the draft C++20 standard.

Requirements

Header: <cstdbool>

Namespace: std

Remarks

Including this header ensures that the names declared using external linkage in the C standard library header are declared in the `std` namespace.

See also

Header files reference
C++ standard library overview
Thread safety in the C++ standard library

<cstddef>

Article • 12/07/2021

Includes the C standard library header <stddef.h> and adds associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C standard library header are declared in the `std` namespace.

> ⓘ Note
>
> <cstddef> includes type **byte** and doesn't include type `wchar_t`.

Syntax

```
C++

#include <cstddef>
```

Namespace and Macros

```
C++

namespace std {
    using ptrdiff_t = see definition;
    using size_t = see definition;
    using max_align_t = see definition;
    using nullptr_t = decltype(nullptr);
}

#define NULL  // an implementation-defined null pointer constant
#define offsetof(type, member-designator)
```

Parameters

ptrdiff_t
An implementation-defined signed integer type that can hold the difference of two subscripts in an array object.

size_t
An implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object.

max_align_t
A POD type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context.

nullptr_t
A synonym for the type of a `nullptr` expression. Although a `nullptr` address can't be taken, the address of another *nullptr_t* object that is an lvalue can be taken.

byte Class

C++

```
enum class byte : unsigned char {};

template <class IntType>
    constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
    constexpr byte operator<<(byte b, IntType shift) noexcept;
    constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
    constexpr byte operator>>(byte b, IntType shift) noexcept;

constexpr byte& operator|=(byte& left, byte right) noexcept;
constexpr byte operator|(byte left, byte right) noexcept;
constexpr byte& operator&=(byte& left, byte right) noexcept;
constexpr byte operator&(byte left, byte right) noexcept;
constexpr byte& operator^=(byte& left, byte right) noexcept;
constexpr byte operator^(byte left, byte right) noexcept;
constexpr byte operator~(byte b) noexcept;

template <class IntType>
    IntType to_integer(byte b) noexcept;
```

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<cstdint>

Article • 12/07/2021

Includes the Standard C library header <stdint.h> and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Syntax

C++

```
#include <cstdint>
```

Types

C++

```
namespace std {
    using int8_t = signed integer type; // optional
    using int16_t = signed integer type; // optional
    using int32_t = signed integer type; // optional
    using int64_t = signed integer type; // optional
    using int_fast8_t = signed integer type;
    using int_fast16_t = signed integer type;
    using int_fast32_t = signed integer type;
    using int_fast64_t = signed integer type;
    using int_least8_t = signed integer type;
    using int_least16_t = signed integer type;
    using int_least32_t = signed integer type;
    using int_least64_t = signed integer type;
    using intmax_t = signed integer type;
    using intptr_t = signed integer type; // optional
    using uint8_t = unsigned integer type; // optional
    using uint16_t = unsigned integer type; // optional
    using uint32_t = unsigned integer type; // optional
    using uint64_t = unsigned integer type; // optional
    using uint_fast8_t = unsigned integer type;
    using uint_fast16_t = unsigned integer type;
    using uint_fast32_t = unsigned integer type;
    using uint_fast64_t = unsigned integer type;
    using uint_least8_t = unsigned integer type;
    using uint_least16_t = unsigned integer type;
    using uint_least32_t = unsigned integer type;
    using uint_least64_t = unsigned integer type;
    using uintmax_t = unsigned integer type;
```

```
    using uintptr_t = unsigned integer type; // optional
}
```

Macros

C++

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX

[U]INT{8 16 32 64 MAX}_C
```

See also

Header Files Reference
C++ Standard Library Overview

<cstdio>

Article • 12/07/2021

Includes the Standard C library header <stdio.h> and adds the associated names to the `std` namespace.

Syntax

```
C++
```

```
#include <cstdio>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

See also

Header Files Reference
C++ Standard Library Overview
Thread Safety in the C++ Standard Library

<cstdlib>

Article • 02/25/2022

Includes the C Standard library header `<stdlib.h>` and adds the associated names to the `std` namespace. Including this header ensures that the names declared using external linkage in the C standard library header are declared in the `std` namespace.

> **Note**
>
> `<stdlib.h>` doesn't include the type **`wchar_t`**.

Requirements

Header: `<cstdlib>`

Namespace: `std`

Namespace and Macros

C++

```
namespace std {
    using size_t = see definition;
    using div_t = see definition;
    using ldiv_t = see definition;
    using lldiv_t = see definition;
}

#define NULL
#define EXIT_FAILURE
#define EXIT_SUCCESS
#define RAND_MAX
#define MB_CUR_MAX
```

Exposition only functions

C++

```
extern "C" using c-atexit-handler = void();
extern "C++" using atexit-handler = void();
```

```
extern "C" using c-compare-pred = int(const void*, const void*);
extern "C++" using compare-pred = int(const void*, const void*);
```

Start and termination functions

Function	Description
_Exit	Terminates program without using destructors or registered functions.
abort	Terminates program without using destructors.
atexit	Registers function for program termination.
exit	Destroys objects with thread and static storage, then returns control.
at_quick_exit	Registers function without arguments for program termination.
quick_exit	Registers function with preserved objects for program termination.
getenv	See C standard library reference.
system	See C standard library reference.

`_Exit`

C++

```
[[noreturn]] void _Exit(int status) noexcept;
```

Remarks

The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()`. The function `_Exit` is signal-safe.

`abort`

C++

```
[[noreturn]] void abort() noexcept;
```

Remarks

The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()`. The function `abort` is signal-safe.

`at_quick_exit`

```
C++
```

```
int at_quick_exit(c-atexit-handler * func) noexcept;
int at_quick_exit(atexit-handler * func) noexcept;
```

Return Value

Zero if the registration succeeds, non-zero if it fails.

Remarks

The `at_quick_exit()` functions register a function *`func`*, which is called without arguments when `quick_exit()` is called. A call to `at_quick_exit()` that doesn't happen before all calls to `quick_exit()` may not succeed. The `at_quick_exit()` functions don't introduce a data race. The order of registration may be indeterminate if `at_quick_exit()` was called from more than one thread. Since `at_quick_exit()` registrations are distinct from the `atexit()` registrations, applications may need to call both registration functions using the same argument. MSVC supports the registration of at least 32 functions.

`atexit`

```
C++
```

```
int atexit(c-atexit-handler * func) noexcept;
int atexit(atexit-handler * func) noexcept;
```

Remarks

The `atexit()` functions register the function pointed to by *`func`* to be called without arguments at normal program termination. A call to `atexit()` that doesn't happen before a call to `exit()` may not succeed. The `atexit()` functions don't introduce a data race.

Return Value

Returns zero if the registration succeeds, nonzero if it fails.

`exit`

```
C++

[[noreturn]] void exit(int status);
```

Remarks

First, objects with thread storage duration and associated with the current thread are destroyed.

Next, objects with static storage duration are destroyed and functions registered by calling `atexit()` are called. Automatic objects aren't destroyed when `exit()` is called. If control leaves a registered function called by `exit()` because the function doesn't provide a handler for a thrown exception, `std::terminate()` is called. A function is called once for every time it's registered. Objects with automatic storage duration are all destroyed in a program whose `main` function contains no automatic objects and executes the call to `exit()`. Control can be transferred directly to such a `main` function by throwing an exception that's caught in `main`.

Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.

Finally, control is returned to the host environment. When *`status`* is zero or `EXIT_SUCCESS`, an implementation-defined form of the status successful termination is returned. MSVC returns a value of zero. If *`status`* is `EXIT_FAILURE`, MSVC returns a value of 3. Otherwise, MSVC returns the *`status`* parameter value.

`getenv`

```
C++

char* getenv(const char* name);
```

`quick_exit`

C++

```
[[noreturn]] void quick_exit(int status) noexcept;
```

Remarks

Generally, functions registered by calls to `at_quick_exit()` are called in the reverse order of their registration. This order doesn't apply to functions registered after other registered functions have already been called. No objects are destroyed when `quick_exit()` is called. If control leaves a registered function called by `quick_exit()` because the function doesn't provide a handler for a thrown exception, `std::terminate()` is called. A function registered via `at_quick_exit()` is invoked by the thread that calls `quick_exit()`, which can be a different thread than the one that registered it. That means registered functions shouldn't rely on the identity of objects that have thread storage duration. After calling registered functions, `quick_exit()` calls `_Exit(status)`. The standard file buffers aren't flushed. The function `quick_exit()` is signal-safe when the functions registered with `at_quick_exit()` are.

`system`

C++

```
int system(const char* string);
```

Memory allocation functions

C++

```
// void* aligned_alloc(size_t alignment, size_t size); // Unsupported in
MSVC
void* calloc(size_t nmemb, size_t size);
void free(void* ptr);
void* malloc(size_t size);
void* realloc(void* ptr, size_t size);
```

Remarks

These functions have the semantics specified in the C standard library. MSVC doesn't support the `aligned_alloc` function. C11 specified `aligned_alloc()` in a way that's

incompatible with the Microsoft implementation of `free()`, namely, that `free()` must be able to handle highly aligned allocations.

Numeric string conversions

C++

```
double atof(const char* nptr);
int atoi(const char* nptr);
long int atol(const char* nptr);
long long int atoll(const char* nptr);
double strtod(const char* nptr, char** endptr);
float strtof(const char* nptr, char** endptr);
long double strtold(const char* nptr, char** endptr);
long int strtol(const char* nptr, char** endptr, int base);
long long int strtoll(const char* nptr, char** endptr, int base);
unsigned long int strtoul(const char* nptr, char** endptr, int base);
unsigned long long int strtoull(const char* nptr, char** endptr, int base);
```

Remarks

These functions have the semantics specified in the C standard library.

Multibyte / wide string and character conversion functions

C++

```
int mblen(const char* s, size_t n);
int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);
```

Remarks

These functions have the semantics specified in the C standard library.

Algorithm functions

C++

```
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
c-compare-pred * compar);
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
compare-pred * compar);
void qsort(void* base, size_t nmemb, size_t size, c-compare-pred * compar);
void qsort(void* base, size_t nmemb, size_t size, compare-pred * compar);
```

Remarks

These functions have the semantics specified in the C standard library.

Low-quality random number generation functions

C++

```
int rand();
void srand(unsigned int seed);
```

Remarks

These functions have the semantics specified in the C standard library.

Absolute values

C++

```
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);
long int labs(long int j);
long long int llabs(long long int j);
```

Remarks

These functions have the semantics specified in the C standard library.

Integer division

```
C++
```

```
div_t div(int numer, int denom);
ldiv_t div(long int numer, long int denom);
lldiv_t div(long long int numer, long long int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Remarks

These functions have the semantics specified in the C standard library.

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cstring>

Article • 12/07/2021

Includes the Standard C library header <string.h> and adds the associated names to the `std` namespace.

Syntax

C++

```
#include <cstring>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants

C++

```
namespace std {
    using size_t = see below;
}

#define NULL
```

Functions

C++

```
void* memcpy(void* s1, const void* s2, size_t n);
void* memmove(void* s1, const void* s2, size_t n);
char* strcpy(char* s1, const char* s2);
char* strncpy(char* s1, const char* s2, size_t n);
char* strcat(char* s1, const char* s2);
char* strncat(char* s1, const char* s2, size_t n);
int memcmp(const void* s1, const void* s2, size_t n);
int strcmp(const char* s1, const char* s2);
int strcoll(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);
```

```
size_t strxfrm(char* s1, const char* s2, size_t n);
const void* memchr(const void* s, int c, size_t n); // see 20.2
void* memchr(void* s, int c, size_t n) // see 20.2
const char* strchr(const char* s, int c) // see 20.2
char* strchr(char* s, int c) // see 20.2
size_t strcspn(const char* s1, const char* s2);
const char* strpbrk(const char* s1, const char* s2) // see 20.2
char* strpbrk(char* s1, const char* s2) // see 20.2
const char* strrchr(const char* s, int c) // see 20.2
char* strrchr(char* s, int c) // see 20.2
size_t strspn(const char* s1, const char* s2);
const char* strstr(const char* s1, const char* s2) // see 20.2
char* strstr(char* s1, const char* s2) // see 20.2
char* strtok(char* s1, const char* s2);
void* memset(void* s, int c, size_t n);
char* strerror(int errnum);
size_t strlen(const char* s);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

`<ctgmath>`;

Article • 08/03/2021

In effect, includes the C++ standard library headers `<complex>` and `<cmath>`, which provide type-generic math macros equivalent to `<tgmath.h>`.

> **Note**
>
> The C standard library `<tgmath.h>` header isn't included by `<ctgmath>` because it's effectively replaced by the C++ overloads in `<complex>` and `<cmath>`. That makes the `<ctgmath>` header redundant. The `<tgmath.h>` header is deprecated in C++. The `<ctgmath>` header is deprecated in C++17 and removed in the draft C++20 standard.

Requirements

Header: `<ctgmath>`

Namespace: `std`

Remarks

The functionality of the C standard library header `<tgmath.h>` is provided by the overloads in `<complex>` and `<cmath>`.

See also

<complex>
<cmath>
Header files reference
C++ standard library overview
Thread safety in the C++ standard library

`<ctime>`

Article • 12/07/2021

Includes the Standard C library header <time.h> and adds the associated names to the `std` namespace.

Syntax

```
C++

#include <ctime>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants

```
C++

#define NULL
#define CLOCKS_PER_SEC
#define TIME_UTC

namespace std {
    using size_t = see below;
    using clock_t = see below ;
    using time_t = see below ;
}
```

Structures

```
C++

struct timespec;
struct tm;
```

Functions

```C++
clock_t clock();
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm* timeptr);
time_t time(time_t* timer);
int timespec_get(timespec* ts, int base);
char* asctime(const struct tm* timeptr);
char* ctime(const time_t* timer);
struct tm* gmtime(const time_t* timer);
struct tm* localtime(const time_t* timer);
size_t strftime(char* s, size_t maxsize, const char* format, const struct
tm* timeptr);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cuchar>

Article • 12/07/2021

Includes the C standard library header <uchar.h> and adds the associated names to the `std` namespace.

Requirements

Header: <cuchar>

Namespace: std

Remarks

This header adds the C standard library types `mbstate_t` and `size_t`, and the functions c16rtomb, c32rtomb, mbrtoc16, and mbrtoc32 to the `std` namespace.

See also

Header files reference
C++ standard library overview
Thread safety in the C++ standard library

<cvt/wbuffer>

Article • 12/07/2021

The header <cvt/wstring> in previous versions of Visual Studio defined the class template wbuffer_convert Class in the stdext::cvt namespace. The header is maintained for backward compatibility. New code should use the version of the class that is defined in <locale> in the std namespace

Syntax

```
C++

#include <cvt/wbuffer>
```

See also

Header Files Reference

`<cvt/wstring>`

Article • 12/07/2021

The header `<cvt/wstring>` in previous versions of Visual Studio defined the class template wstring_convert Class in the stdext::cvt namespace. The header is maintained for backward compatibility. New code should use the version of the class that is defined in <locale> in the `std` namespace

Syntax

```
C++

#include <cvt/wstring>
```

Requirements

See also

Header Files Reference

<cwchar>

Article • 12/07/2021

Includes the Standard C library header <wchar.h> and adds the associated names to the `std` namespace.

Syntax

```
C++

#include <cwchar>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants

```
C++

namespace std {
    using size_t = see below;
    using mbstate_t = see below ;
    using wint_t = see below ;
}

#define NULL see below
#define WCHAR_MAX see below
#define WCHAR_MIN see below
#define WEOF see below
```

Structs

```
C++

struct tm;
```

Functions

C++

```
int fwprintf(FILE* stream, const wchar_t* format, ...);
int fwscanf(FILE* stream, const wchar_t* format, ...);
int swprintf(wchar_t* s, size_t n, const wchar_t* format, ...);
int swscanf(const wchar_t* s, const wchar_t* format, ...);
int vfwprintf(FILE* stream, const wchar_t* format, va_list arg);
int vfwscanf(FILE* stream, const wchar_t* format, va_list arg);
int vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
int vwprintf(const wchar_t* format, va_list arg);
int vwscanf(const wchar_t* format, va_list arg);
int wprintf(const wchar_t* format, ...);
int wscanf(const wchar_t* format, ...);
wint_t fgetwc(FILE* stream);
wchar_t* fgetws(wchar_t* s, int n, FILE* stream);
wint_t fputwc(wchar_t c, FILE* stream);
int fputws(const wchar_t* s, FILE* stream);
int fwide(FILE* stream, int mode);
wint_t getwc(FILE* stream);
wint_t getwchar();
wint_t putwc(wchar_t c, FILE* stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE* stream);
double wcstod(const wchar_t* nptr, wchar_t** endptr);
float wcstof(const wchar_t* nptr, wchar_t** endptr);
long double wcstold(const wchar_t* nptr, wchar_t** endptr);
long int wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
long long int wcstoll(const wchar_t* nptr, wchar_t** endptr, int base);
unsigned long int wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);
unsigned long long int wcstoull(const wchar_t* nptr, wchar_t** endptr, int
base);
wchar_t* wcscpy(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemcpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemmove(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wcscat(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncat(wchar_t* s1, const wchar_t* s2, size_t n);
int wcscmp(const wchar_t* s1, const wchar_t* s2);
int wcscoll(const wchar_t* s1, const wchar_t* s2);
int wcsncmp(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(wchar_t* s1, const wchar_t* s2, size_t n);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c) // see 20.2
wchar_t* wcschr(wchar_t* s, wchar_t c) // see 20.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2) // see 20.2
wchar_t* wcspbrk(wchar_t* s1, const wchar_t* s2) // see 20.2
const wchar_t* wcsrchr(const wchar_t* s, wchar_t c) // see 20.2
wchar_t* wcsrchr(wchar_t* s, wchar_t c) // see 20.2
size_t wcsspn(const wchar_t* s1, const wchar_t* s2);
```

```
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2) // see 20.2
wchar_t* wcsstr(wchar_t* s1, const wchar_t* s2) // see 20.2
wchar_t* wcstok(wchar_t* s1, const wchar_t* s2, wchar_t** ptr);
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n) // see 20.2
wchar_t* wmemchr(wchar_t* s, wchar_t c, size_t n) // see 20.2
size_t wcslen(const wchar_t* s);
wchar_t* wmemset(wchar_t* s, wchar_t c, size_t n);
size_t wcsftime(wchar_t* s, size_t maxsize, const wchar_t* format, const
struct tm* timeptr);
wint_t btowc(int c);
int wctob(wint_t c);
// 24.5.6, multibyte / wide string and character conversion functions
int mbsinit(const mbstate_t* ps);
size_t mbrlen(const char* s, size_t n, mbstate_t* ps);
size_t mbrtowc(wchar_t* pwc, const char* s, size_t n, mbstate_t* ps);
size_t wcrtomb(char* s, wchar_t wc, mbstate_t* ps);
size_t mbsrtowcs(wchar_t* dst, const char** src, size_t len, mbstate_t* ps);
size_t wcsrtombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<cwctype>

Article • 12/07/2021

Includes the Standard C library header <wctype.h> and adds the associated names to the `std` namespace.

Syntax

```
C++

#include <cwctype>
```

Remarks

Including this header ensures that the names declared using external linkage in the Standard C library header are declared in the `std` namespace.

Constants

```
C++

namespace std {
    using wint_t = see below ;
    using wctrans_t = see below ;
    using wctype_t = see below ;
}

#define WEOF see below
```

Functions

```
C++

int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
```

```
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char* property);
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char* property);
```

See also

Header Files Reference

C++ Standard Library Overview

Thread Safety in the C++ Standard Library

<deque>

Article • 12/07/2021

Defines the container class template deque and several supporting templates.

Requirements

Header: <deque>

> ⓘ Note
>
> The <deque> library also uses the `#include <initializer_list>` statement.

Members

Operators

Name	Description
operator!=	Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.
operator<	Tests if the deque object on the left side of the operator is less than the deque object on the right side.
operator<=	Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.
operator==	Tests if the deque object on the left side of the operator is equal to the deque object on the right side.
operator>	Tests if the deque object on the left side of the operator is greater than the deque object on the right side.
operator>=	Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

Functions

Name	Description

Name	Description
swap	Exchanges the elements of two deques.

Classes

Name	Description
deque Class	A class template of sequence containers that arrange elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<deque>` functions

Article • 12/07/2021

swap

Exchanges the elements of two deques.

```
C++
```

```
void swap(
    deque<Type, Allocator>& left,
    deque<Type, Allocator>& right,);
```

Parameters

left

An object of type `deque`.

right

An object of type `deque`.

Example

See the example for deque::swap.

`<deque>` operators

Article • 12/07/2021

operator!=

Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.

```
C++
```

```
bool operator!=(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque objects are not equal; **`false`** if the deque objects are equal.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deque objects are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// deque_op_ne.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c2.push_back( 2 );

   if ( c1 != c2 )
      cout << "The deques are not equal." << endl;
   else
      cout << "The deques are equal." << endl;
}
```

Output

```
The deques are not equal.
```

`operator<`

Tests if the deque object on the left side of the operator is less than the deque object on the right side.

C++

```
bool operator<(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque on the left side of the operator is less than and not equal to the deque on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// deque_op_lt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 < c2 )
      cout << "Deque c1 is less than deque c2." << endl;
   else
      cout << "Deque c1 is not less than deque c2." << endl;
}
```

Output

```
Deque c1 is less than deque c2.
```

operator<=

Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.

C++

```
bool operator<=(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque on the left side of the operator is less than or equal to the deque on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// deque_op_le.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 <= c2 )
      cout << "Deque c1 is less than or equal to deque c2." << endl;
   else
      cout << "Deque c1 is greater than deque c2." << endl;
}
```

Output

```
Deque c1 is less than or equal to deque c2.
```

operator==

Tests if the deque object on the left side of the operator is equal to the deque object on the right side.

C++

```
bool operator==(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque on the left side of the operator is equal to the deque on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. Two deques are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// deque_op_eq.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c2.push_back( 1 );

   if ( c1 == c2 )
      cout << "The deques are equal." << endl;
   else
      cout << "The deques are not equal." << endl;

   c1.push_back( 1 );
   if ( c1 == c2 )
      cout << "The deques are equal." << endl;
   else
      cout << "The deques are not equal." << endl;
}
```

Output

```
The deques are equal.
The deques are not equal.
```

`operator>`

Tests if the deque object on the left side of the operator is greater than the deque object on the right side.

C++

```
bool operator>(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque on the left side of the operator is greater than the deque on the right side of the operator; otherwise `false`.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// deque_op_gt.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 > c2 )
      cout << "Deque c1 is greater than deque c2." << endl;
   else
      cout << "Deque c1 is not greater than deque c2." << endl;
}
```

Output

```
Deque c1 is greater than deque c2.
```

`operator>=`

Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

C++

```
bool operator>=(const deque<Type, Allocator>& left, const deque<Type,
Allocator>& right);
```

Parameters

left
An object of type `deque`.

right
An object of type `deque`.

Return Value

`true` if the deque on the left side of the operator is greater than or equal to the deque on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between deque objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// deque_op_ge.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2;

   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );
```

```
   if ( c1 >= c2 )
      cout << "Deque c1 is greater than or equal to deque c2." << endl;
   else
      cout << "Deque c1 is less than deque c2." << endl;
}
```

Output

```
Deque c1 is greater than or equal to deque c2.
```

`deque` Class

Article • 10/17/2022

Arranges elements of a given type in a linear arrangement and, like a vector, enables fast random access to any element, and efficient insertion and deletion at the back of the container. However, unlike a vector, the `deque` class also supports efficient insertion and deletion at the front of the container.

Syntax

unstlib

```
template <class Type, class Allocator =allocator<Type>>
class deque
```

Parameters

`Type`

The element data type to be stored in the `deque`.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the `deque`'s allocation and deallocation of memory. This argument is optional, and the default value is **`allocator<Type>`**.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the list container is superior when efficient insertions and deletions (in constant time) at any location within the sequence is at a premium. Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

`Deque` reallocation occurs when a member function must insert or erase elements of the sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by begin and end become invalid.
- If an element is inserted at the first position of the `deque`, then all iterators, but no references, that designate existing elements become invalid.
- If an element is inserted at the end of the `deque`, then end and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at the front of the `deque`, only that iterator and references to the erased element become invalid.
- If the last element is erased from the end of the `deque`, only that iterator to the final element and references to the erased element become invalid.

Otherwise, inserting or erasing an element invalidates all iterators and references.

Members

Constructors

Name	Description
deque	Constructs a `deque`. Several constructors are provided to set up the contents of the new `deque` in different ways: empty; loaded with a specified number of empty elements; contents moved or copied from another `deque`; contents copied or moved by using an iterator; and one element copied into the `deque` `count` times. Some of the constructors enable using a custom `allocator` to create elements.

Typedefs

Name	Description
allocator_type	A type that represents the `allocator` class for the `deque` object.
const_iterator	A type that provides a random-access iterator that can access and read elements in the `deque` as **`const`**
const_pointer	A type that provides a pointer to an element in a `deque` as a `const.`
const_reference	A type that provides a reference to an element in a `deque` for reading and other operations as a `const.`

Name	Description
const_reverse_iterator	A type that provides a random-access iterator that can access and read elements in the `deque` as **`const`**. The `deque` is viewed in reverse. For more information, see reverse_iterator Class
difference_type	A type that provides the difference between two random-access iterators that refer to elements in the same `deque`.
iterator	A type that provides a random-access iterator that can read or modify any element in a `deque`.
pointer	A type that provides a pointer to an element in a `deque`.
reference	A type that provides a reference to an element stored in a `deque`.
reverse_iterator	A type that provides a random-access iterator that can read or modify an element in a `deque`. The `deque` is viewed in reverse order.
size_type	A type that counts the number of elements in a `deque`.
value_type	A type that represents the data type stored in a `deque`.

Functions

Name	Description
assign	Erases elements from a `deque` and copies a new sequence of elements to the target `deque`.
at	Returns a reference to the element at a specified location in the `deque`.
back	Returns a reference to the last element of the `deque`.
begin	Returns a random-access iterator addressing the first element in the `deque`.
cbegin	Returns a `const` iterator to the first element in the `deque`.
cend	Returns a random-access **`const`** iterator that points just beyond the end of the `deque`.
clear	Erases all the elements of a `deque`.
crbegin	Returns a random-access `const` iterator to the first element in a `deque` viewed in reverse order.
crend	Returns a random-access `const` iterator to the first element in a `deque` viewed in reverse order.

Name	Description
emplace	Inserts an element constructed in place into the `deque` at a specified position.
emplace_back	Adds an element constructed in place to the end of the `deque`.
emplace_front	Adds an element constructed in place to the start of the `deque`.
empty	Returns `true` if the `deque` contains zero elements, and `false` if it contains one or more elements.
end	Returns a random-access iterator that points just beyond the end of the `deque`.
erase	Removes an element or a range of elements in a `deque` from specified positions.
front	Returns a reference to the first element in a `deque`.
get_allocator	Returns a copy of the `allocator` object that is used to construct the `deque`.
insert	Inserts an element, several elements, or a range of elements into the `deque` at a specified position.
max_size	Returns the maximum possible length of the `deque`.
pop_back	Erases the element at the end of the `deque`.
pop_front	Erases the element at the start of the `deque`.
push_back	Adds an element to the end of the `deque`.
push_front	Adds an element to the start of the `deque`.
rbegin	Returns a random-access iterator to the first element in a reversed `deque`.
rend	Returns a random-access iterator that points just beyond the last element in a reversed `deque`.
resize	Specifies a new size for a `deque`.
shrink_to_fit	Discards excess capacity.
size	Returns the number of elements in the `deque`.
swap	Exchanges the elements of two `deque`s.

Operators

Name	Description
operator[]	Returns a reference to the `deque` element at a specified position.

Name	Description
operator=	Replaces the elements of the `deque` with a copy of another `deque`.

`allocator_type`

A type that represents the allocator class for the `deque` object.

```
C++
```

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

Example

See the example for get_allocator.

`assign`

Erases elements from a `deque` and copies a new set of elements to the target `deque`.

```
C++
```

```
template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);

void assign(
    size_type Count,
    const Type& Val);

void assign(initializer_list<Type> IList);
```

Parameters

`First`

Position of the first element in the range of elements to be copied from the argument

`deque`.

`Last`
Position of the first element beyond the range of elements to be copied from the argument `deque`.

`Count`
The number of copies of an element being inserted into the `deque`.

`Val`
The value of the element being inserted into the `deque`.

`IList`
The `initializer_list` being inserted into the `deque`.

Remarks

After any existing elements in the target `deque` are erased, `assign` either inserts a specified range of elements from the original `deque` or from some other `deque` into the target `deque`, or inserts copies of a new element of a specified value into the target `deque`.

Example

C++

```
// deque_assign.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
#include <initializer_list>

int main()
{
    using namespace std;
    deque <int> c1, c2;
    deque <int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    deque<int> d1{ 1, 2, 3, 4 };
```

```
    initializer_list<int> iList{ 5, 6, 7, 8 };
    d1.assign(iList);

    cout << "d1 = ";
    for (int i : d1)
        cout << i;
    cout << endl;

    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(7, 4);
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;
}
```

Output

```
d1 = 5678c1 =102030c1 =5060c1 =4444444
```

at

Returns a reference to the element at a specified location in the `deque`.

C++

```
reference at(size_type pos);

const_reference at(size_type pos) const;
```

Parameters

pos

The subscript (or position number) of the element to reference in the `deque`.

Return Value

If `pos` is greater than the size of the `deque`, `at` throws an exception.

Remarks

If the return value of `at` is assigned to a `const_reference`, the `deque` object can't be modified. If the return value of `at` is assigned to a `reference`, the `deque` object can be modified.

Example

C++

```
// deque_at.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const int& i = c1.at( 0 );
   int& j = c1.at( 1 );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

back

Returns a reference to the last element of the `deque`.

C++

```
reference back();
const_reference back() const;
```

Return Value

The last element of the `deque`. If the `deque` is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the `deque` object can't be modified. If the return value of `back` is assigned to a `reference`, the `deque` object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty `deque`. See Checked Iterators for more information.

Example

C++

```
// deque_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

`begin`

Returns an iterator addressing the first element in the `deque`.

```
C++
```

```
const_iterator begin() const;
iterator begin();
```

Return Value

A random-access iterator addressing the first element in the `deque` or to the location succeeding an empty `deque`.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the `deque` object can't be modified. If the return value of `begin` is assigned to an `iterator`, the `deque` object can be modified.

Example

```
C++
```

```
// deque_begin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::const_iterator c1_cIter;

   c1.push_back( 1 );
   c1.push_back( 2 );

   c1_Iter = c1.begin( );
   cout << "The first element of c1 is " << *c1_Iter << endl;

*c1_Iter = 20;
   c1_Iter = c1.begin( );
   cout << "The first element of c1 is now " << *c1_Iter << endl;

   // The following line would be an error because iterator is const
   // *c1_cIter = 200;
}
```

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

cbegin

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`clear`

Erases all the elements of a `deque`.

```
C++
```

```
void clear();
```

Example

C++

```
// deque_clear.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "The size of the deque is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of the deque after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the deque is initially 3
The size of the deque after clearing is 0
```

`const_iterator`

A type that provides a random-access iterator that can access and read a **`const`** element in the `deque`.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

Example

See the example for back.

`const_pointer`

Provides a pointer to a **`const`** element in a `deque`.

C++

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element. An iterator is more commonly used to access a `deque` element.

`const_reference`

A type that provides a reference to a **`const`** element stored in a `deque` for reading and performing **`const`** operations.

C++

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` can't be used to modify the value of an element.

Example

C++

```
// deque_const_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
```

```
    const deque <int> c2 = c1;
    const int &i = c2.front( );
    const int &j = c2.back( );
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

    // The following line would cause an error as c2 is const
    // c2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

`const_reverse_iterator`

A type that provides a random-access iterator that can read any `const` element in the `deque`.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the `deque` in reverse.

Example

See the example for rbegin for an example of how to declare and use an iterator.

`crbegin`

Returns a `const` iterator to the first element in a reversed `deque`.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A `const` reverse random-access iterator addressing the first element in a reversed deque or addressing what had been the last element in the unreversed `deque`.

Remarks

With the return value of `crbegin`, the `deque` object can't be modified.

Example

C++

```
// deque_crbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;
   deque <int>::iterator v1_Iter;
   deque <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   v1_Iter = v1.begin( );
   cout << "The first element of deque is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.crbegin( );
   cout << "The first element of the reversed deque is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of deque is 1.
The first element of the reversed deque is 2.
```

`crend`

Returns a `const` iterator that addresses the location succeeding the last element in a reversed `deque`.

```
C++
```

```
const_reverse_iterator crend() const;
```

Return Value

A `const` reverse random-access iterator that addresses the location succeeding the last element in a reversed deque (the location that had preceded the first element in the unreversed `deque`).

Remarks

`crend` is used with a reversed `deque` just as array::cend is used with a `deque`.

With the return value of `crend` (suitably decremented), the `deque` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `deque`.

The value returned by `crend` shouldn't be dereferenced.

Example

```
C++
```

```
// deque_crend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;
   deque <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

deque

Constructs a `deque` of a specific size, or with elements of a specific value, or with a specific allocator, or as a copy of all or part of some other `deque`.

C++

```
deque();

explicit deque(const Allocator& Al);
explicit deque(size_type Count);
deque(size_type Count, const Type& Val);

deque(
    size_type Count,
    const Type& Val,
    const Allocator& Al);

deque(const deque& Right);

template <class InputIterator>
deque(InputIterator First,  InputIterator Last);

template <class InputIterator>
deque(
   InputIterator First,
   InputIterator Last,
   const Allocator& Al);

deque(initializer_list<value_type> IList, const Allocator& Al);
```

Parameters

`Al`

The allocator class to use with this object.

`Count`

The number of elements in the constructed `deque`.

`Val`

The value of the elements in the constructed `deque`.

Right

The deque of which the constructed deque is to be a copy.

First

Position of the first element in the range of elements to be copied.

Last

Position of the first element beyond the range of elements to be copied.

IList

The initializer_list to be copied.

Remarks

All constructors store an allocator object (*Al*) and initialize the deque.

The first two constructors specify an empty initial deque; the second one also specifies the allocator type (_Al) to be used.

The third constructor specifies a repetition of a specified number (count) of elements of the default value for class Type.

The fourth and fifth constructors specify a repetition of (*Count*) elements of value val.

The sixth constructor specifies a copy of the deque *Right*.

The seventh and eighth constructors copy the range [First, Last) of a deque.

The seventh constructor moves the deque *Right*.

The eighth constructor copies the contents of an initializer_list.

None of the constructors perform any interim reallocations.

Example

C++

```
/ compile with: /EHsc
#include <deque>
#include <iostream>
#include <forward_list>

int main()
{
```

```
    using namespace std;

    forward_list<int> f1{ 1, 2, 3, 4 };

    f1.insert_after(f1.begin(), { 5, 6, 7, 8 });

    deque <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter,
c6_Iter;

    // Create an empty deque c0
    deque <int> c0;

    // Create a deque c1 with 3 elements of default value 0
    deque <int> c1(3);

    // Create a deque c2 with 5 elements of value 2
    deque <int> c2(5, 2);

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque <int> c3(3, 1, c2.get_allocator());

    // Create a copy, deque c4, of deque c2
    deque <int> c4(c2);

    // Create a deque c5 by copying the range c4[ first,  last)
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    deque <int> c5(c4.begin(), c4_Iter);

    // Create a deque c6 by copying the range c4[ first,  last) and
    // c2 with the allocator of deque
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    c4_Iter++;
    deque <int> c6(c4.begin(), c4_Iter, c2.get_allocator());

    // Create a deque c8 by copying the contents of an initializer_list
    // using brace initialization
    deque<int> c8({ 1, 2, 3, 4 });

    initializer_list<int> iList{ 5, 6, 7, 8 };
    deque<int> c9( iList);

    cout << "c1 = ";
    for (int i : c1)
        cout << i << " ";
    cout << endl;

    cout << "c2 = ";
    for (int i : c2)
        cout << i << " ";
    cout << endl;
```

```
    cout << "c3 = ";
    for (int i : c3)
        cout << i << " ";
    cout << endl;

    cout << "c4 = ";
    for (int i : c4)
        cout << i << " ";
    cout << endl;

    cout << "c5 = ";
    for (int i : c5)
        cout << i << " ";
    cout << endl;

    cout << "c6 = ";
    for (int i : c6)
        cout << i << " ";
    cout << endl;

    // Move deque c6 to deque c7
    deque <int> c7(move(c6));
    deque <int>::iterator c7_Iter;

    cout << "c7 =";
    for (int i : c7)
        cout << i << " ";
    cout << endl;

    cout << "c8 = ";
    for (int i : c8)
        cout << i << " ";
    cout << endl;

    cout << "c9 = ";
    for (int i : c9)
        cout << i << " ";
    cout << endl;

    int x = 3;
}
// deque_deque.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
   deque <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter,
c6_Iter;

    // Create an empty deque c0
    deque <int> c0;
```

```
    // Create a deque c1 with 3 elements of default value 0
    deque <int> c1( 3 );

    // Create a deque c2 with 5 elements of value 2
    deque <int> c2( 5, 2 );

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque <int> c3( 3, 1, c2.get_allocator( ) );

    // Create a copy, deque c4, of deque c2
    deque <int> c4( c2 );

    // Create a deque c5 by copying the range c4[ first,  last)
    c4_Iter = c4.begin( );
    c4_Iter++;
    c4_Iter++;
    deque <int> c5( c4.begin( ), c4_Iter );

    // Create a deque c6 by copying the range c4[ first,  last) and
    // c2 with the allocator of deque
    c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   c4_Iter++;
   deque <int> c6( c4.begin( ), c4_Iter, c2.get_allocator( ) );

    // Create a deque c8 by copying the contents of an initializer_list
    // using brace initialization
    deque<int> c8({ 1, 2, 3, 4 });

        initializer_list<int> iList{ 5, 6, 7, 8 };
    deque<int> c9( iList);

    cout << "c1 = ";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
        cout << *c1_Iter << " ";
    cout << endl;

    cout << "c2 = ";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
        cout << *c2_Iter << " ";
    cout << endl;

    cout << "c3 = ";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
        cout << *c3_Iter << " ";
    cout << endl;

    cout << "c4 = ";
    for ( c4_Iter = c4.begin( ); c4_Iter != c4.end( ); c4_Iter++ )
        cout << *c4_Iter << " ";
    cout << endl;
```

```
    cout << "c5 = ";
    for ( c5_Iter = c5.begin( ); c5_Iter != c5.end( ); c5_Iter++ )
        cout << *c5_Iter << " ";
    cout << endl;

    cout << "c6 = ";
    for ( c6_Iter = c6.begin( ); c6_Iter != c6.end( ); c6_Iter++ )
        cout << *c6_Iter << " ";
    cout << endl;

    // Move deque c6 to deque c7
    deque <int> c7( move(c6) );
    deque <int>::iterator c7_Iter;

    cout << "c7 =" ;
    for ( c7_Iter = c7.begin( ) ; c7_Iter != c7.end( ) ; c7_Iter++ )
        cout << " " << *c7_Iter;
    cout << endl;

    cout << "c8 = ";
    for (int i : c8)
        cout << i << " ";
    cout << endl;

    cout << "c9 = ";
    or (int i : c9)
        cout << i << " ";
    cout << endl;
}
```

`difference_type`

A type that provides the difference between two iterators that refer to elements within the same `deque`.

C++

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

A `difference_type` can also be described as the number of elements between two pointers.

Example

C++

```
// deque_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <deque>
#include <algorithm>

int main( )
{
   using namespace std;

   deque <int> c1;
   deque <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

   deque <int>::difference_type df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

emplace

Inserts an element constructed in place into the `deque` at a specified position.

C++

```
iterator emplace(
    const_iterator _Where,
    Type&& val);
```

Parameters

_Where

The position in the deque where the first element is inserted.

val

The value of the element being inserted into the `deque`.

Return Value

The function returns an iterator that points to the position where the new element was inserted into the `deque`.

Remarks

Any insertion operation can be expensive, see `deque` for a discussion of `deque` performance.

Example

C++

```
// deque_emplace.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;
   deque <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

// initialize a deque of deques by moving v1
   deque < deque <int> > vv1;
```

```
   vv1.emplace( vv1.begin(), move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      {
      cout << "vv1[0] =";
      for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
         cout << " " << *Iter;
      cout << endl;
      }
}
```

Output

```
v1 = 10 20 30
vv1[0] = 10 20 30
```

`emplace_back`

Adds an element constructed in place to the end of the `deque`.

C++

```
void emplace_back(Type&& val);
```

Parameters

`val`

The element added to the end of the deque.

Example

C++

```
// deque_emplace_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;

   v1.push_back( 1 );
   if ( v1.size( ) != 0 )
      cout << "Last element: " << v1.back( ) << endl;
```

```
   v1.push_back( 2 );
   if ( v1.size( ) != 0 )
      cout << "New last element: " << v1.back( ) << endl;

// initialize a deque of deques by moving v1
   deque < deque <int> > vv1;

   vv1.emplace_back( move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      cout << "Moved last element: " << vv1[0].back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved last element: 2
```

`emplace_front`

Adds an element constructed in place to the end of the `deque`.

C++

```
void emplace_front(Type&& val);
```

Parameters

`val`

The element added to the beginning of the deque.

Example

C++

```
// deque_emplace_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;
```

```
   v1.push_back( 1 );
   if ( v1.size( ) != 0 )
      cout << "Last element: " << v1.back( ) << endl;

   v1.push_back( 2 );
   if ( v1.size( ) != 0 )
      cout << "New last element: " << v1.back( ) << endl;

// initialize a deque of deques by moving v1
   deque < deque <int> > vv1;

   vv1.emplace_front( move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      cout << "Moved last element: " << vv1[0].back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved last element: 2
```

`empty`

Tests if a `deque` is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the `deque` is empty; **`false`** if the `deque` isn't empty.

Example

C++

```
// deque_empty.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
```

```
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The deque is empty." << endl;
   else
      cout << "The deque is not empty." << endl;
}
```

Output

```
The deque is not empty.
```

end

Returns an iterator that addresses the location succeeding the last element in a `deque`.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

A random-access iterator that addresses the location succeeding the last element in a `deque`. If the `deque` is empty, then `deque::end == deque::begin`.

Remarks

`end` is used to test whether an iterator has reached the end of its `deque`.

Example

C++

```
// deque_end.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    deque <int> c1;
    deque <int>::iterator c1_Iter;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1_Iter = c1.end( );
    c1_Iter--;
    cout << "The last integer of c1 is " << *c1_Iter << endl;

    c1_Iter--;
    *c1_Iter = 400;
    cout << "The new next-to-last integer of c1 is " << *c1_Iter << endl;

    // If a const iterator had been declared instead with the line:
    // deque <int>::const_iterator c1_Iter;
    // an error would have resulted when inserting the 400

    cout << "The deque is now:";
    for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
       cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The deque is now: 10 400 30
```

erase

Removes an element or a range of elements in a `deque` from specified positions.

C++

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);
```

Parameters

`_Where`

Position of the element to be removed from the `deque`.

first

Position of the first element removed from the `deque`.

last

Position just beyond the last element removed from the `deque`.

Return Value

A random-access iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the `deque` if no such element exists.

Remarks

`erase` never throws an exception.

Example

C++

```
// deque_erase.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial deque is: ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the deque becomes:  ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
   cout << "After erasing all elements but the first, deque becomes: ";
```

```
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
}
```

Output

```
The initial deque is: 10 20 30 40 50
After erasing the first element, the deque becomes:  20 30 40 50
After erasing all elements but the first, deque becomes: 20
```

`front`

Returns a reference to the first element in a `deque`.

C++

```
reference front();

const_reference front() const;
```

Return Value

If the `deque` is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the `deque` object can't be modified. If the return value of `front` is assigned to a `reference`, the `deque` object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty `deque`. See Checked Iterators for more information.

Example

C++

```
// deque_front.cpp
// compile with: /EHsc
#include <deque>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.front( );
   const int& ii = c1.front( );

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The second integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 11
```

`get_allocator`

Returns a copy of the allocator object used to construct the `deque`.

C++

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the `deque`.

Remarks

Allocators for the `deque` class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// deque_get_allocator.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   deque <int> c1;
   deque <int, allocator<int> > c2 = deque <int, allocator<int> >(
allocator<int>( ) );

   // c3 will use the same allocator class as c1
   deque <int> c3( c1.get_allocator( ) );

   deque <int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

insert

Inserts an element or a number of elements or a range of elements into the `deque` at a specified position.

C++

```
iterator insert(
    const_iterator Where,
    const Type& Val);

iterator insert(
    const_iterator Where,
    Type&& Val);

void insert(
    iterator Where,
    size_type Count,
    const Type& Val);

template <class InputIterator>
void insert(
    iterator Where,
    InputIterator First,
    InputIterator Last);

iterator insert(
    iterator Where,initializer_list<Type>
IList);
```

Parameters

Where

The position in the target `deque` where the first element is inserted.

Val

The value of the element being inserted into the `deque`.

Count

The number of elements being inserted into the `deque`.

First

The position of the first element in the range of elements in the argument `deque` to be copied.

Last

The position of the first element beyond the range of elements in the argument `deque` to be copied.

IList

The `initializer_list` of elements to insert.

Return Value

The first two insert functions return an iterator that points to the position where the new element was inserted into the `deque`.

Remarks

Any insertion operation can be expensive.

`iterator`

A type that provides a random-access iterator that can read or modify any element in a `deque`.

C++

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin.

`max_size`

Returns the maximum length of the `deque`.

```
C++
```

```
size_type max_size() const;
```

Return Value

The maximum possible length of the `deque`.

Example

```
C++
```

```
// deque_max_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   i = c1.max_size( );
   cout << "The maximum possible length of the deque is " << i << "." <<
endl;
}
```

`operator[]`

Returns a reference to the `deque` element at a specified position.

C++

```
reference operator[](size_type pos);

const_reference operator[](size_type pos) const;
```

Parameters

pos
The position of the `deque` element to be referenced.

Return Value

A reference to the element whose position is specified in the argument. If the position specified is greater than the size of the `deque`, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the `deque` object can't be modified. If the return value of `operator[]` is assigned to a `reference`, the `deque` object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the `deque`. See Checked Iterators for more information.

Example

C++

```
// deque_op_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   cout << "The first integer of c1 is " << c1[0] << endl;
   int& i = c1[1];
```

```
    cout << "The second integer of c1 is " << i << endl;
}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 20
```

`operator=`

Replaces the elements of this `deque` using the elements from another `deque`.

C++

```
deque& operator=(const deque& right);

deque& operator=(deque&& right);
```

Parameters

`right`
The `deque` that provides the new content.

Remarks

The first override copies elements to this `deque` from *`right`*, the source of the assignment. The second override moves elements to this `deque` from *`right`*.

Elements that are contained in this `deque` before the operator executes are removed.

Example

C++

```
// deque_operator_as.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
using namespace std;

typedef deque<int> MyDeque;

template<typename MyDeque> struct S;
```

```
template<typename MyDeque> struct S<MyDeque&> {
  static void show( MyDeque& d ) {
    MyDeque::const_iterator iter;
    for (iter = d.cbegin(); iter != d.cend(); iter++)
       cout << *iter << " ";
    cout << endl;
  }
};

template<typename MyDeque> struct S<MyDeque&&> {
  static void show( MyDeque&& d ) {
    MyDeque::const_iterator iter;
    for (iter = d.cbegin(); iter != d.cend(); iter++)
       cout << *iter << " ";
cout << " via unnamed rvalue reference " << endl;
  }
};

int main( )
{
   MyDeque d1, d2;

   d1.push_back(10);
   d1.push_back(20);
   d1.push_back(30);
   d1.push_back(40);
   d1.push_back(50);

   cout << "d1 = " ;
   S<MyDeque&>::show( d1 );

   d2 = d1;
   cout << "d2 = ";
   S<MyDeque&>::show( d2 );

   cout << "     ";
   S<MyDeque&&>::show ( move< MyDeque& > (d1) );
}
```

`pointer`

Provides a pointer to an element in a deque.

unstlib

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element. An iterator is more commonly used to access a `deque` element.

pop_back

Deletes the element at the end of the `deque`.

C++

```
void pop_back();
```

Remarks

The last element must not be empty. `pop_back` never throws an exception.

Example

C++

```
// deque_pop_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the deque, the "
      "last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the deque, the last element is: 1
```

pop_front

Deletes the element at the beginning of the `deque`.

C++

```
void pop_front();
```

Remarks

The first element must not be empty. `pop_front` never throws an exception.

Example

C++

```
// deque_pop_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the "
      "deque, the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the `deque`, the first
element is: 2
```

push_back

Adds an element to the end of the `deque`.

```
C++
```

```
void push_back(const Type& val);

void push_back(Type&& val);
```

Parameters

`val`
The element added to the end of the `deque`.

Remarks

If an exception is thrown, the `deque` is left unaltered and the exception is rethrown.

`push_front`

Adds an element to the beginning of the `deque`.

```
C++
```

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

`val`
The element added to the beginning of the `deque`.

Remarks

If an exception is thrown, the `deque` is left unaltered and the exception is rethrown.

Example

```
C++
```

```
// deque_push_front.cpp
// compile with: /EHsc
```

```cpp
#include <deque>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;

// move initialize a deque of strings
   deque <string> c2;
   string str("a");

   c2.push_front( move( str ) );
   cout << "Moved first element: " << c2.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
Moved first element: a
```

`rbegin`

Returns an iterator to the first element in a reversed `deque`.

C++

```cpp
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse random-access iterator addressing the first element in a reversed `deque` or addressing what had been the last element in the unreversed `deque`.

Remarks

`rbegin` is used with a reversed `deque` just as begin is used with a `deque`.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the `deque` object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the `deque` object can be modified.

`rbegin` can be used to iterate through a `deque` backwards.

Example

C++

```
// deque_rbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rbegin( );
   cout << "Last element in the deque is " << *c1_rIter << "." << endl;

   cout << "The deque contains the elements: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << "in that order.";
   cout << endl;

   // rbegin can be used to iterate through a deque in reverse order
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;
```

```
   c1_rIter = c1.rbegin( );
   *c1_rIter = 40;  // This would have caused an error if a
                    // const_reverse iterator had been declared as
                    // noted above
   cout << "Last element in deque is now " << *c1_rIter << "." << endl;
}
```

Output

```
Last element in the deque is 30.
The deque contains the elements: 10 20 30 in that order.
The reversed deque is: 30 20 10
Last element in deque is now 40.
```

reference

A type that provides a reference to an element stored in a `deque`.

C++

```
typedef typename Allocator::reference reference;
```

Example

C++

```
// deque_reference.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

rend

Returns an iterator that addresses the location succeeding the last element in a reversed `deque`.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed `deque` (the location that had preceded the first element in the unreversed `deque`).

Remarks

`rend` is used with a reversed `deque` just as end is used with a `deque`.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the `deque` object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, the `deque` object can be modified.

`rend` can be used to test whether a reverse iterator has reached the end of its `deque`.

The value returned by `rend` shouldn't be dereferenced.

Example

C++

```
// deque_rend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
```

```
{
   using namespace std;

   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;
   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rend( );
   c1_rIter --; // Decrementing a reverse iterator moves it forward
                // in the deque (to point to the first element here)
   cout << "The first element in the deque is: " << *c1_rIter << endl;

   cout << "The deque is: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << endl;

   // rend can be used to test if an iteration is through all of
   // the elements of a reversed deque
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;

   c1_rIter = c1.rend( );
   c1_rIter--; // Decrementing the reverse iterator moves it backward
               // in the reversed deque (to the last element here)
   *c1_rIter = 40; // This modification of the last element would
                   // have caused an error if a const_reverse
                   // iterator had been declared (as noted above)
   cout << "The modified reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;
}
```

Output

```
The first element in the deque is: 10
The deque is: 10 20 30
The reversed deque is: 30 20 10
The modified reversed deque is: 30 20 40
```

`resize`

Specifies a new size for a `deque`.

```
C++
```

```
void resize(size_type _Newsize);

void resize(size_type _Newsize, Type val);
```

Parameters

`_Newsize`
The new size of the `deque`.

`val`
The value of the new elements to be added to the `deque` if the new size is larger that the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the `deque`'s `size` is less than *`_Newsize`*, elements are added to the `deque` until it reaches the size *`_Newsize`*.

If the `deque`'s `size` is larger than *`_Newsize`*, the elements closest to the end of the `deque` are deleted until the `deque` reaches the size *`_Newsize`*.

If the present size of the `deque` is the same as *`_Newsize`*, no action is taken.

size reflects the current size of the `deque`.

Example

```
C++
```

```
// deque_resize.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    deque <int> c1;

    c1.push_back( 10 );
    c1.push_back( 20 );
    c1.push_back( 30 );

    c1.resize( 4,40 );
    cout << "The size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is " << c1.back( ) << endl;

    c1.resize( 5 );
    cout << "The size of c1 is now: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;

    c1.resize( 2 );
    cout << "The reduced size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is: 4
The value of the last element is 40
The size of c1 is now: 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

`reverse_iterator`

A type that provides a random-access iterator that can read or modify an element in a reversed `deque`.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the `deque`.

Example

See the example for rbegin.

`shrink_to_fit`

Discards excess capacity.

C++

```
void shrink_to_fit();
```

Remarks

There's no portable way to determine if `shrink_to_fit` reduces the storage used by a deque.

Example

C++

```
// deque_shrink_to_fit.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> v1;
   //deque <int>::iterator Iter;

   v1.push_back( 1 );
   v1.push_back( 2 );
   cout << "Current size of v1 = "
      << v1.size( ) << endl;
   v1.shrink_to_fit();
   cout << "Current size of v1 = "
      << v1.size( ) << endl;
}
```

Output

```
Current size of v1 = 1
Current size of v1 = 1
```

`size`

Returns the number of elements in the `deque`.

C++

```
size_type size() const;
```

Return Value

The current length of the `deque`.

Example

C++

```
// deque_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   c1.push_back( 1 );
   i = c1.size( );
   cout << "The deque length is " << i << "." << endl;

   c1.push_back( 2 );
   i = c1.size( );
   cout << "The deque length is now " << i << "." << endl;
}
```

Output

```
The deque length is 1.
The deque length is now 2.
```

`size_type`

A type that counts the number of elements in a `deque`.

C++

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for size.

swap

Exchanges the elements of two deques.

C++

```
void swap(deque<Type, Allocator>& right);

friend void swap(deque<Type, Allocator>& left, deque<Type, Allocator>&
right) template <class Type, class Allocator>
void swap(deque<Type, Allocator>& left, deque<Type, Allocator>& right);
```

Parameters

right

The deque providing the elements to be swapped, or the deque whose elements are to be exchanged with those of the deque *left*.

left

A deque whose elements are to be exchanged with those of the deque *right*.

Example

C++

```
// deque_swap.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2, c3;
   deque <int>::iterator c1_Iter;

   c1.push_back( 1 );
```

```
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap( c1,c3 );

   cout << "After swapping with c3, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap<>(c1, c2);
   cout << "After swapping with c2, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
The original deque c1 is: 1 2 3
After swapping with c2, deque c1 is: 10 20
After swapping with c3, deque c1 is: 100
After swapping with c2, deque c1 is: 1 2 3
```

value_type

A type that represents the data type stored in a `deque`.

C++

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Type`.

Example

C++

```
// deque_value_type.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
int main( )
{
   using namespace std;
   deque<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

<exception>

Article • 12/07/2021

Defines several types and functions related to the handling of exceptions. Exception handling is used in situations in which the system can recover from an error. It provides a means for control to be returned from a function to the program. The objective of incorporating exception handling is to increase the program's robustness while providing a way to recover from an error in an orderly fashion.

Requirements

Header: <exception>

Namespace: std

Members

Typedefs

Name	Description
exception_ptr	A type that describes a pointer to an exception.
terminate_handler	A type that describes a pointer to a function suitable for use as a `terminate_handler`.
unexpected_handler	A type that describes a pointer to a function suitable for use as an `unexpected_handler`.

Functions

Name	Description
current_exception	Obtains a pointer to the current exception.
get_terminate	Obtains the current `terminate_handler` function.
get_unexpected	Obtains the current `unexpected_handler` function.
make_exception_ptr	Creates an `exception_ptr` object that holds a copy of an exception.
rethrow_exception	Throws an exception passed as a parameter.

Name	Description
rethrow_if_nested	Casts and throws exception if nested.
set_terminate	Establishes a new `terminate_handler` to be called at the termination of the program.
set_unexpected	Establishes a new `unexpected_handler` to be when an unexpected exception is encountered.
terminate	Calls a terminate handler.
throw_with_nested	Throws exception if nested.
uncaught_exception	Returns `true` only if a thrown exception is being currently processed.
unexpected	Calls an unexpected handler.

Classes

Name	Description
bad_exception Class	The class describes an exception that can be thrown from an `unexpected_handler`.
exception Class	The class serves as the base class for all exceptions thrown by certain expressions and by the C++ Standard Library.
nested_exception Class	The class describes an exception that can be captured and stored for later use.

See also

Header Files Reference
Thread Safety in the C++ Standard Library

`<exception>` functions

Article • 12/07/2021

current_exception

Obtains a smart pointer to the current exception.

```
C++

exception_ptr current_exception();
```

Return Value

An exception_ptr object pointing to the current exception.

Remarks

Call the `current_exception` function in a catch block. If an exception is in flight and the catch block can catch the exception, the `current_exception` function returns an `exception_ptr` object that references the exception. Otherwise, the function returns a null `exception_ptr` object.

The `current_exception` function captures the exception that is in flight regardless of whether the **`catch`** statement specifies an exception-declaration statement.

The destructor for the current exception is called at the end of the **`catch`** block if you do not rethrow the exception. However, even if you call the `current_exception` function in the destructor, the function returns an `exception_ptr` object that references the current exception.

Successive calls to the `current_exception` function return `exception_ptr` objects that refer to different copies of the current exception. Consequently, the objects compare as unequal because they refer to different copies, even though the copies have the same binary value.

make_exception_ptr

Creates an exception_ptr object that holds a copy of an exception.

```
C++
```

```
template <class E>
    exception_ptr make_exception_ptr(E Except);
```

Parameters

Except
The class with the exception to copy. Usually, you specify an exception class object as the argument to the `make_exception_ptr` function, although any class object can be the argument.

Return Value

An exception_ptr object pointing to a copy of the current exception for *Except*.

Remarks

Calling the `make_exception_ptr` function is equivalent to throwing a C++ exception, catching it in a catch block, and then calling the current_exception function to return an `exception_ptr` object that references the exception. The Microsoft implementation of the `make_exception_ptr` function is more efficient than throwing and then catching an exception.

An application typically does not require the `make_exception_ptr` function, and we discourage its use.

rethrow_exception

Throws an exception passed as a parameter.

```
C++
```

```
void rethrow_exception(exception_ptr P);
```

Parameters

P
The caught exception to re-throw. If *P* is a null exception_ptr, the function throws std::bad_exception.

Remarks

After you store a caught exception in an `exception_ptr` object, the primary thread can process the object. In your primary thread, call the `rethrow_exception` function together with the `exception_ptr` object as its argument. The `rethrow_exception` function extracts the exception from the `exception_ptr` object and then throws the exception in the context of the primary thread.

get_terminate

Obtains the current `terminate_handler` function.

```
C++
```

```
terminate_handler get_terminate();
```

set_terminate

Establishes a new `terminate_handler` to be called at the termination of the program.

```
C++
```

```
terminate_handler set_terminate(terminate_handler fnew) throw();
```

Parameters

fnew
The function to be called at termination.

Return Value

The address of the previous function that used to be called at termination.

Remarks

The function establishes a new terminate_handler as the function * *fnew*. Thus, *fnew* must not be a null pointer. The function returns the address of the previous terminate handler.

Example

C++

```
// exception_set_terminate.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>

using namespace std;

void termfunction()
{
    cout << "My terminate function called." << endl;
    abort();
}

int main()
{
    terminate_handler oldHandler = set_terminate(termfunction);

    // Throwing an unhandled exception would also terminate the program
    // or we could explicitly call terminate();

    //throw bad_alloc();
    terminate();
}
```

get_unexpected

Obtains the current `unexpected_handler` function.

C++

```
unexpected_handler get_unexpected();
```

rethrow_if_nested

C++

```
template <class E>
    void rethrow_if_nested(const E& e);
```

Remarks

If not a polymorphic class type, or if `nested_exception` is inaccessible or ambiguous, there is no effect. Otherwise, performs a dynamic cast.

set_unexpected

Establishes a new `unexpected_handler` to be when an unexpected exception is encountered.

C++

```
unexpected_handler set_unexpected(unexpected_handler fnew) throw();
```

Parameters

fnew
The function to be called when an unexpected exception is encountered.

Return Value

The address of the previous `unexpected_handler`.

Remarks

fnew must not be a null pointer.

The C++ Standard requires that `unexpected` is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The following example calls `unexpected` directly, which then calls the `unexpected_handler`.

Example

C++

```
// exception_set_unexpected.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>

using namespace std;

void uefunction()
```

```
{
    cout << "My unhandled exception function called." << endl;
    terminate(); // this is what unexpected() calls by default
}

int main()
{
    unexpected_handler oldHandler = set_unexpected(uefunction);

    unexpected(); // library function to force calling the
                  // current unexpected handler
}
```

terminate

Calls a terminate handler.

C++

```
void terminate();
```

Remarks

The function calls a terminate handler, a function of type **`void`**. If `terminate` is called directly by the program, the terminate handler is the one most recently set by a call to set_terminate. If `terminate` is called for any of several other reasons during evaluation of a throw expression, the terminate handler is the one in effect immediately after evaluating the throw expression.

A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls `abort`.

Example

See set_unexpected for an example of the use of `terminate`.

throw_with_nested

C++

```
template <class T> [[noreturn]]
    void throw_with_nested(T&& t);
```

Remarks

Throws exception with nested exceptions.

uncaught_exception

Returns `true` only if a thrown exception is being currently processed.

C++

```
bool uncaught_exception();
```

Return Value

Returns `true` after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling unexpected as a result of the throw expression. In particular, `uncaught_exception` will return `true` when called from a destructor that is being invoked during an exception unwind. On devices, `uncaught_exception` is only supported on Windows CE 5.00 and higher versions, including Windows Mobile 2005 platforms.

Example

C++

```
// exception_uncaught_exception.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <string>

class Test
{
public:
   Test( std::string msg ) : m_msg( msg )
   {
      std::cout << "In Test::Test(\"" << m_msg << "\")" << std::endl;
   }
   ~Test( )
   {
      std::cout << "In Test::~Test(\"" << m_msg << "\")" << std::endl
         << "        std::uncaught_exception( ) = "
         << std::uncaught_exception( )
         << std::endl;
   }
```

```
private:
    std::string m_msg;
};

// uncaught_exception will be true in the destructor
// for the object created inside the try block because
// the destructor is being called as part of the unwind.

int main( void )
   {
      Test t1( "outside try block" );
      try
      {
         Test t2( "inside try block" );
         throw 1;
      }
      catch (...) {
   }
}
```

Output

```
In Test::Test("outside try block")
In Test::Test("inside try block")
In Test::~Test("inside try block")
        std::uncaught_exception( ) = 1
In Test::~Test("outside try block")
        std::uncaught_exception( ) = 0
```

unexpected

Calls the unexpected handler.

C++

```
void unexpected();
```

Remarks

The C++ Standard requires that `unexpected` is called when a function throws an exception that is not on its throw list. The current implementation does not support this. The example calls `unexpected` directly, which calls the unexpected handler.

The function calls an unexpected handler, a function of type **`void`**. If `unexpected` is called directly by the program, the unexpected handler is the one most recently set by a call to set_unexpected.

An unexpected handler may not return to its caller. It may terminate execution by:

- Throwing an object of a type listed in the exception specification or an object of any type if the unexpected handler is called directly by the program.
- Throwing an object of type bad_exception.
- Calling terminate, `abort` or `exit`.

At program startup, the unexpected handler is a function that calls terminate.

Example

See set_unexpected for an example of the use of `unexpected`.

`<exception>` typedefs

Article • 12/07/2021

exception_ptr

A type that describes a pointer to an exception.

```
C++

typedef unspecified exception_ptr;
```

Remarks

An unspecified internal class that is used to implement the `exception_ptr` type.

Use an `exception_ptr` object to reference the current exception or an instance of a user-specified exception. In the Microsoft implementation, an exception is represented by an EXCEPTION_RECORD structure. Each `exception_ptr` object includes an exception reference field that points to a copy of the `EXCEPTION_RECORD` structure that represents the exception.

When you declare an `exception_ptr` variable, the variable is not associated with any exception. That is, its exception reference field is NULL. Such an `exception_ptr` object is called a *null exception_ptr*.

Use the `current_exception` or `make_exception_ptr` function to assign an exception to an `exception_ptr` object. When you assign an exception to an `exception_ptr` variable, the variable's exception reference field points to a copy of the exception. If there is insufficient memory to copy the exception, the exception reference field points to a copy of a std::bad_alloc exception. If the `current_exception` or `make_exception_ptr` function cannot copy the exception for any other reason, the function calls the `terminate` CRT function to exit the current process.

Despite its name, an `exception_ptr` object is not itself a pointer. It does not obey pointer semantics and cannot be used with the pointer member access (`->`) or indirection (*) operators. The `exception_ptr` object has no public data members or member functions.

Comparisons:

You can use the equal (`==`) and not-equal (`!=`) operators to compare two `exception_ptr` objects. The operators do not compare the binary value (bit pattern) of the `EXCEPTION_RECORD` structures that represent the exceptions. Instead, the operators compare the addresses in the exception reference field of the `exception_ptr` objects. Consequently, a null `exception_ptr` and the NULL value compare as equal.

terminate_handler

The type describes a pointer to a function suitable for use as a `terminate_handler`.

```
C++
```

```
typedef void (*terminate_handler)();
```

Remarks

The type describes a pointer to a function suitable for use as a terminate handler.

Example

See set_terminate for an example of the use of `terminate_handler`.

unexpected_handler

The type describes a pointer to a function suitable for use as an `unexpected_handler`.

```
C++
```

```
typedef void (*unexpected_handler)();
```

Example

See set_unexpected for an example of the use of `unexpected_handler`.

bad_exception Class

Article • 08/03/2021

The class describes an exception that can be thrown from an unexpected handler.

Syntax

C++

```
class bad_exception : public exception {};

bad_exception();
bad_exception(const bad_exception&);
bad_exception& operator=(const bad_exception&);
const char* what() const override;
```

Remarks

unexpected will throw a `bad_exception` instead of terminating or instead of calling another function specified with set_unexpected if `bad_exception` is included in the throw list of a function.

The value returned by `what` is an implementation-defined C string. None of the member functions throw any exceptions.

For a list of members inherited by the `bad_exception` class, see exception Class.

Example

See set_unexpected for an example of the use of unexpected throwing a `bad_exception`.

`exception` Class

Article • 03/01/2022

The class serves as the base class for all exceptions thrown by certain expressions and by the C++ Standard Library.

Syntax

C++

```
class exception {
   public:
   exception();
   exception(const char* const &message);
   exception(const char* const &message, int);
   exception(const exception &right);
   exception& operator=(const exception &right);
   virtual ~exception();
   virtual const char *what() const;
};
```

Remarks

Specifically, this base class is the root of the standard exception classes defined in <stdexcept>. The C string value returned by `what` is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string. None of the member functions throw any exceptions.

The **`int`** parameter allows you to specify that no memory should be allocated. The value of the **`int`** is ignored.

> ⓘ **Note**
>
> The constructors `exception(const char* const &message)` and `exception(const char* const &message, int)` are Microsoft extensions to the C++ Standard Library.

Example

For examples of the use of the standard exception classes that inherit from the `exception` class, see any of the classes defined in <stdexcept>.

<execution>

Article • 09/13/2024

Describes the execution policies for parallel algorithms.

Syntax

C++

```
namespace std {
    template<class T> inline constexpr bool is_execution_policy_v =
is_execution_policy<T>::value;
}
namespace std::execution {
    inline constexpr sequenced_policy seq { unspecified };
    inline constexpr parallel_policy par { unspecified };
    inline constexpr parallel_unsequenced_policy par_unseq { unspecified };
}
```

Classes and structs

Expand table

Name	Description
is_execution_policy Struct	Detects execution policies to exclude certain function signatures from otherwise ambiguous overload resolution participation.
parallel_policy class	Used to disambiguate parallel algorithm overloading. Indicates that a parallel algorithm's execution may be parallelized.
parallel_unsequenced_policy class	Used as a unique type to disambiguate parallel algorithm overloading. Indicates that a parallel algorithm's execution may be parallelized and vectorized.
sequenced_policy class	Used as a unique type to disambiguate parallel algorithm overloading. Specifies that a parallel algorithm's execution may not be parallelized.

Microsoft specific

Parallel algorithms execute on an unspecified number of threads and divide the work into an unspecified number of data partitioning "chunks." The Windows thread pool manages the number of threads. The implementation tries to make use of the available

logical processors, which corresponds to the number of hardware threads that can execute simultaneously.

Specifying `parallel_policy` or `parallel_unsequenced_policy` causes standard library algorithms to run in parallel using the Windows Thread Pool. The number of concurrent threads, and thus the number of "chunks" for data partitioning, is limited to 500 threads because that's the default number of thread pool threads. For more information, see Thread Pools.

Before Windows 11 and Windows Server 2022, applications were limited by default to a single processor group having at most 64 logical processors. This limited the number of concurrently executing threads to 64. For more information, see Processor Groups.

Starting with Windows 11 and Windows Server 2022, processes and their threads have processor affinities that by default span all processors in the system and across multiple groups on machines with more than 64 processors. The limit on the number of concurrent threads is now the total number of logical processors in the system.

Requirements

Header: `<execution>`

Namespace: `std`

See also

Header files reference
Thread safety in the C++ standard library
C++ standard library reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`filesystem`

Article • 12/06/2021

Include the header `<filesystem>` for access to classes and functions that manipulate and retrieve information about paths, files, and directories.

Syntax

C++

```
#include <filesystem> // C++17 standard header file name
#include <experimental/filesystem> // Header file for pre-standard
implementation
using namespace std::experimental::filesystem::v1;
```

> **ⓘ Important**
>
> At the release of Visual Studio 2017, the `<filesystem>` header was not yet a C++ standard. C++ in Visual Studio 2017 RTW implements the final draft standard, found in **ISO/IEC JTC 1/SC 22/WG 21 N4100**. Visual Studio 2017 version 15.7 and later supports the new C++17 `<filesystem>` standard. This is a completely new implementation, incompatible with the previous `std::experimental` version. It was made necessary by symlink support, bug fixes, and changes in standard-required behavior. In Visual Studio 2019 version 16.3 and later, including `<filesystem>` provides only the new `std::filesystem`. Including `<experimental/filesystem>` provides only the old experimental implementation. The experimental implementation will be removed in the next ABI-breaking release of the libraries.

This header supports file systems for one of two broad classes of host operating systems: Microsoft Windows and POSIX.

While most functionality is common to both operating systems, this document identifies where differences occur. For example:

- Windows supports multiple root names, such as `c:` or `\\network_name`. A file system consists of a forest of trees, each with its own root directory, such as `c:\` or `\\network_name\`, and each with its own current directory, for completing a relative pathname (one that's not an absolute pathname).

- POSIX supports a single tree, with no root name, the single root directory `/`, and a single current directory.

Another significant difference is the native representation of pathnames:

- Windows uses a null-terminated sequence of **`wchar_t`**, encoded as UTF-16 (one or more elements for each character).
- POSIX uses a null-terminated sequence of **`char`**, encoded as UTF-8 (one or more elements for each character).
- An object of class `path` stores the pathname in native form, but supports easy conversion between this stored form and several external forms:
  - A null-terminated sequence of **`char`**, encoded as favored by the operating system.
  - A null-terminated sequence of **`char`**, encoded as UTF-8.
  - A null-terminated sequence of **`wchar_t`**, encoded as favored by the operating system.
  - A null-terminated sequence of **`char16_t`**, encoded as UTF-16.
  - A null-terminated sequence of **`char32_t`**, encoded as UTF-32.

  Interconversions between these representations are mediated, as needed, by the use of one or more `codecvt` facets. If no specific locale object is specified, these facets are obtained from the global locale.

Another difference is the detail with which each operating system lets you specify file or directory access permissions:

- Windows records whether a file is read-only or writable, an attribute that has no meaning for directories.
- POSIX records whether a file can be read, written, or executed (scanned, if a directory). And, whether each operation is allowed for the owner, the owner's group, or for everybody, plus a few other permissions.

Common to both systems is the structure imposed on a pathname once you get past the root name. For the pathname `c:/abc/xyz/def.ext`:

- The root name is `c:`.

- The root directory is `/`.
- The root path is `c:/`.
- The relative path is `abc/xyz/def.ext`.
- The parent path is `c:/abc/xyz`.
- The filename is `def.ext`.
- The stem is `def`.
- The extension is `.ext`.

A minor difference is the preferred separator between the sequence of directories in a pathname. Both operating systems let you write a forward slash `/`, but in some contexts Windows prefers a backslash `\`. The implementation stores its preferred separator in the data member `preferred_separator` in `path`.

Finally, `path` objects have an important feature: You can use them wherever a filename argument is required in the classes defined in the header <fstream>.

For more information and code examples, see File system navigation (C++).

Members

Classes

Name	Description
directory_entry class	Describes an object that is returned by a `directory_iterator` or a `recursive_directory_iterator` and contains a `path`.
directory_iterator class	Describes an input iterator that sequences through the file names in a file-system directory.
filesystem_error class	A base class for exceptions that are thrown to report a low-level system overflow.
path class	Defines a class that stores an object of template type `String` that is suitable for use as a file name.
recursive_directory_iterator class	Describes an input iterator that sequences through the file names in a file-system directory. The iterator can also descend into subdirectories.

Name	Description
file_status class	Wraps a `file_type`.

Structs

Name	Description
space_info structure	Holds information about a volume.

Functions

<filesystem> functions

Operators

<filesystem> operators

Enumerations

Name	Description
copy_options	An enumeration that is used with copy_file and determines behavior if a destination file already exists.
directory_options	An enumeration that specifies options for directory iterators.
file_type	An enumeration for file types.
perm_options	Enumerates options for the `permissions` function.
perms	A bitmask type used to convey permissions and options to permissions

See also

Header files reference

`<filesystem>` operators

Article • 12/07/2021

The operators perform a lexical comparison of two paths as strings. Use the `equivalent` function to determine whether two paths (for example a relative path and an absolute path) refer to the same file or directory on disk.

For more information, see File System Navigation (C++).

operator==

```
C++
```

```
bool operator==(const path& left, const path& right) noexcept;
```

The function returns left.native() == right.native().

operator!=

```
C++
```

```
bool operator!=(const path& left, const path& right) noexcept;
```

The function returns !(left == right).

operator<

```
C++
```

```
bool operator<(const path& left, const path& right) noexcept;
```

The function returns left.native() < right.native().

operator<=

```
C++
```

```
bool operator<=(const path& left, const path& right) noexcept;
```

The function returns !(right < left).

operator>

```
C++
```

```
bool operator>(const path& left, const path& right) noexcept;
```

The function returns right < left.

operator>=

```
C++
```

```
bool operator>=(const path& left, const path& right) noexcept;
```

The function returns !(left < right).

operator/

```
C++
```

```
path operator/(const path& left, const path& right);
```

The function executes:

```
C++
```

```
basic_string<Elem, Traits> str;
path ans = left;
return (ans /= right);
```

operator<<

```
C++
```

```
template <class Elem, class Traits>
basic_ostream<Elem, Traits>& operator<<(basic_ostream<Elem, Traits>& os,
const path& pval);
```

The function returns os << pval.string<Elem, Traits>().

operator>>

```
C++
```

```
template <class Elem, class Traits>
basic_istream<Elem, Traits>& operator<<(basic_istream<Elem, Traits>& is,
const path& pval);
```

The function executes:

```
C++
```

```
basic_string<Elem, Traits> str;
is>> str;
pval = str;
return (is);
```

`<filesystem>` functions

Article • 02/08/2022

These free functions in the <filesystem> header do modifying and query operations on paths, files, symlinks, directories, and volumes. For more information and code examples, see File System Navigation (C++).

absolute

C++

```
path absolute(const path& pval, const path& base = current_path());
```

The function returns the absolute pathname corresponding to *pval* relative to the pathname `base`:

1. If `pval.has_root_name() && pval.has_root_directory()` the function returns *pval*.
2. If `pval.has_root_name() && !pval.has_root_directory()` the function returns `pval.root_name()` / `absolute(base).root_directory()` / `absolute(base).relative_path()` / `pval.relative_path()`.
3. If `!pval.has_root_name() && pval.has_root_directory()` the function returns `absolute(base).root_name()` / *pval*.
4. If `!pval.has_root_name() && !pval.has_root_directory()` the function returns `absolute(base)` / *pval*.

begin

C++

```
const directory_iterator& begin(const directory_iterator& iter) noexcept;
const recursive_directory_iterator&
    begin(const recursive_directory_iterator& iter) noexcept;
```

Both functions return *iter*.

canonical

C++

```
path canonical(const path& pval, const path& base = current_path());
path canonical(const path& pval, error_code& ec);
path canonical(const path& pval, const path& base, error_code& ec);
```

The functions all form an absolute pathname `pabs = absolute(pval, base)` (or `pabs = absolute(pval)` for the overload with no base parameter), then reduce it to a canonical form in the following sequence of steps:

1. Every path component `X` for which `is_symlink(X)` is **`true`** is replaced by `read_symlink(X)`.
2. Every path component `.` (dot is the current directory established by previous path components) is removed.
3. Every pair of path components `X/..` (dot-dot is the parent directory established by previous path components) is removed.

The function then returns `pabs`.

copy

C++

```
void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
void copy(const path& from, const path& to, copy_options opts);
void copy(const path& from, const path& to, copy_options opts, error_code&
ec) noexcept;
```

The functions all possibly copy or link one or more files at *from* to *to* under control of *`opts`*, which is taken as `copy_options::none` for the overloads with no *`opts`* parameter. *`opts`* shall contain at most one of:

- `skip_existing`, `overwrite_existing`, or `update_existing`
- `copy_symlinks` or `skip_symlinks`
- `directories_only`, `create_symlinks`, or `create_hard_links`

The functions first determine the file_status values `f` for *from* and `t` for *to*:

- if `opts & (copy_options::create_symlinks | copy_options::skip_symlinks)`, by calling `symlink_status`
- otherwise, by calling `status`
- Otherwise report an error.

If `!exists(f) || equivalent(f, t) || is_other(f) || is_other(t) || is_directory(f)&& is_regular_file(t)`, they then report an error (and do nothing else).

Otherwise, if `is_symlink(f)` then:

- If `options & copy_options::skip_symlinks`, then do nothing.
- Otherwise, if `!exists(t)&& options & copy_options::copy_symlinks`, then `copy_symlink(from, to, opts)`.
- Otherwise, report an error.

Otherwise, if `is_regular_file(f)`, then:

- If `opts & copy_options::directories_only`, then do nothing.
- Otherwise, if `opts & copy_options::create_symlinks`, then `create_symlink(to, from)`.
- Otherwise, if `opts & copy_options::create_hard_links`, then `create_hard_link(to, from)`.
- Otherwise, if `is_directory(f)`, then `copy_file(from, to` / `from.filename(), opts)`.
- Otherwise, `copy_file(from, to, opts)`.

Otherwise, if `is_directory(f) && (opts & copy_options::recursive || !opts)`, then:

C++

```
if (!exists(t))
{  // copy directory contents recursively
    create_directory(to, from, ec);

    for (directory_iterator next(from), end; ec == error_code() && next != end; ++next)
    {
        copy(next->path(), to / next->path().filename(), opts, ec);
    }
}
```

Otherwise, do nothing.

`copy_file`

```
C++
```

```
bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec) noexcept;
bool copy_file(const path& from, const path& to, copy_options opts);
bool copy_file(const path& from, const path& to, copy_options opts,
error_code& ec) noexcept;
```

The functions all possibly copy the file at *from* to *to* under control of `opts`, which is taken as `copy_options::none` for the overloads with no `opts` parameter. `opts` shall contain at most one of `skip_existing`, `overwrite_existing`, or `update_existing`.

If `exists(to) && !(opts & (copy_options::skip_existing | copy_options::overwrite_existing | copy_options::update_existing))`, then report as an error that the file already exists.

Otherwise, if `!exists(to) || opts & copy_options::overwrite_existing || opts & copy_options::update_existing&& last_write_time(to) < last_write_time(from) || !(opts & (copy_options::skip_existing | copy_options::overwrite_existing | copy_options:update_existing))`, then attempt to copy the contents and attributes of the file *from* to the file *to*. Report as an error if the copy attempt fails.

The functions return **`true`** if the copy is attempted and succeeds, otherwise **`false`**.

`copy_symlink`

```
C++
```

```
void copy_symlink(const path& from, const path& to);
void copy_symlink(const path& from, const path& to, error_code& ec)
noexcept;
```

If `is_directory(from)`, the function calls `create_directory_symlink(from, to)`. Otherwise, it calls `create_symlink(from, to)`.

`create_directories`

```
C++

bool create_directories(const path& pval);
bool create_directories(const path& pval, error_code& ec) noexcept;
```

For a pathname such as `a\/b\/c`, the function creates directories `a` and `a\/b` as needed so that it can create the directory `a\/b\/c` as needed. It returns **`true`** only if it actually creates the directory *`pval`*.

`create_directory`

```
C++

bool create_directory(const path& pval);

bool create_directory(const path& pval, error_code& ec) noexcept;
bool create_directory(const path& pval, const path& attr);
bool create_directory(const path& pval, const path& attr, error_code& ec)
noexcept;
```

The function creates the directory *`pval`* as needed. It returns `true` only if it actually creates the directory *`pval`*, in which case it copies permissions from the existing file *`attr`*, or uses `perms::all` for the overloads with no *`attr`* parameter.

`create_directory_symlink`

```
C++

void create_directory_symlink(const path& to, const path& link);
void create_directory_symlink(const path& to, const path& link, error_code&
ec) noexcept;
```

The function creates link as a symlink to the directory *to*.

`create_hard_link`

```
C++

void create_hard_link(const path& to,  const path& link);
void create_hard_link(const path& to, const path& link, error_code& ec)
noexcept;
```

The function creates link as a hard link to the directory or file *to*.

`create_symlink`

```
C++
```

```
void create_symlink(const path& to, const path& link);

void create_symlink(const path& to, const path& link, error_code& ec)
noexcept;
```

The function creates `link` as a symlink to the file *to*.

`current_path`

```
C++
```

```
path current_path();
path current_path(error_code& ec);
void current_path(const path& pval);
void current_path(const path& pval, error_code& ec) noexcept;
```

The functions with no parameter `pval` return the pathname for the current directory. The remaining functions set the current directory to `pval`.

`end`

```
C++
```

```
directory_iterator& end(const directory_iterator& iter) noexcept;
recursive_directory_iterator& end(const recursive_directory_iterator& iter)
noexcept;
```

The first function returns `directory_iterator()` and the second function returns `recursive_directory_iterator()`

`equivalent`

```
C++
```

```
bool equivalent(const path& left, const path& right);
bool equivalent(const path& left, const path& right, error_code& ec)
noexcept;
```

The functions return `true` only if *left* and *right* choose the same filesystem entity.

exists

C++

```
bool exists(file_status stat) noexcept;
bool exists(const path& pval);
bool exists(const path& pval, error_code& ec) noexcept;
```

The first function returns `status_known && stat.type() != file_not_found`. The second and third functions return `exists(status(pval))`.

file_size

C++

```
uintmax_t file_size(const path& pval);
uintmax_t file_size(const path& pval, error_code& ec) noexcept;
```

The functions return the size in bytes of the file chosen by *`pval`*, if `exists(pval) && is_regular_file(pval)` and the file size can be determined. Otherwise they report an error and return `uintmax_t(-1)`.

hard_link_count

C++

```
uintmax_t hard_link_count(const path& pval);
uintmax_t hard_link_count(const path& pval, error_code& ec) noexcept;
```

The function returns the number of hard links for *`pval`*, or -1 if an error occurs.

hash_value

```C++
size_t hash_value(const path& pval) noexcept;
```

The function returns a hash value for `pval.native()`.

is_block_file

```C++
bool is_block_file(file_status stat) noexcept;
bool is_block_file(const path& pval);
bool is_block_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::block`. The remaining functions return `is_block_file(status(pval))`.

is_character_file

```C++
bool is_character_file(file_status stat) noexcept;
bool is_character_file(const path& pval);
bool is_character_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::character`. The remaining functions return `is_character_file(status(pval))`.

is_directory

```C++
bool is_directory(file_status stat) noexcept;
bool is_directory(const path& pval);
bool is_directory(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::directory`. The remaining functions return `is_directory_file(status(pval))`.

is_empty

```
C++
```

```
bool is_empty(file_status stat) noexcept;
bool is_empty(const path& pval);
bool is_empty(const path& pval, error_code& ec) noexcept;
```

If `is_directory(pval)`, then the function returns `directory_iterator(pval) == directory_iterator()`; otherwise it returns `file_size(pval) == 0`.

is_fifo

```
C++
```

```
bool is_fifo(file_status stat) noexcept;
bool is_fifo(const path& pval);
bool is_fifo(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::fifo`. The remaining functions return `is_fifo(status(pval))`.

is_other

```
C++
```

```
bool is_other(file_status stat) noexcept;
bool is_other(const path& pval);
bool is_other(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::other`. The remaining functions return `is_other(status(pval))`.

is_regular_file

```
C++
```

```
bool is_regular_file(file_status stat) noexcept;
bool is_regular_file(const path& pval);
bool is_regular_file(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::regular`. The remaining functions return `is_regular_file(status(pval))`.

`is_socket`

```
C++
```

```
bool is_socket(file_status stat) noexcept;
bool is_socket(const path& pval);
bool is_socket(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::socket`. The remaining functions return `is_socket(status(pval))`.

`is_symlink`

```
C++
```

```
bool is_symlink(file_status stat) noexcept;
bool is_symlink(const path& pval);
bool is_symlink(const path& pval, error_code& ec) noexcept;
```

The first function returns `stat.type() == file_type::symlink`. The remaining functions return `is_symlink(status(pval))`.

`last_write_time`

```
C++
```

```
file_time_type last_write_time(const path& pval);
file_time_type last_write_time(const path& pval, error_code& ec) noexcept;
void last_write_time(const path& pval, file_time_type new_time);
void last_write_time(const path& pval, file_time_type new_time, error_code&
ec) noexcept;
```

The first two functions return the time of last data modification for *`pval`*, or `file_time_type(-1)` if an error occurs. The last two functions set the time of last data modification for *`pval`* to *`new_time`*.

`permissions`

```
C++
```

```
void permissions(const path& pval, perms mask);
void permissions(const path& pval, perms mask, error_code& ec) noexcept;
```

The functions set the permissions for the pathname chosen by *pval* to `mask & perms::mask` under control of `perms & (perms::add_perms | perms::remove_perms)`. *mask* shall contain at most one of `perms::add_perms` and `perms::remove_perms`.

If `mask & perms::add_perms`, the functions set the permissions to `status(pval).permissions() | mask & perms::mask`. Otherwise, if `mask & perms::remove_perms`, the functions set the permissions to `status(pval).permissions() & ~(mask & perms::mask)`. Otherwise, the functions set the permissions to `mask & perms::mask`.

proximate

C++

```
path proximate(const path& p, error_code& ec);
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);
```

read_symlink

C++

```
path read_symlink(const path& pval);
path read_symlink(const path& pval, error_code& ec);
```

The functions report an error and return `path()` if `!is_symlink(pval)`. Otherwise, the functions return an object of type `path` containing the symbolic link.

relative

C++

```
path relative(const path& p, error_code& ec);
path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);
```

`remove`

```
C++
```

```
bool remove(const path& pval);
bool remove(const path& pval, error_code& ec) noexcept;
```

The functions return **`true`** only if `exists(symlink_status(pval))` and the file is successfully removed. A symlink is itself removed, not the file it chooses.

`remove_all`

```
C++
```

```
uintmax_t remove_all(const path& pval);
uintmax_t remove_all(const path& pval, error_code& ec) noexcept;
```

If *`pval`* is a directory, the functions recursively remove all directory entries, then the entry itself. Otherwise, the functions call `remove`. They return a count of all elements successfully removed.

`rename`

```
C++
```

```
void rename(const path& from, const path& to);
void rename(const path& from, const path& to, error_code& ec) noexcept;
```

The functions rename *from* to *to*. A symlink is itself renamed, not the file it chooses.

`resize_file`

```
C++
```

```
void resize(const path& pval, uintmax_t size);
void resize(const path& pval, uintmax_t size, error_code& ec) noexcept;
```

The functions alter the size of a file such that `file_size(pval) == size`

space

```
C++
```

```
space_info space(const path& pval);
space_info space(const path& pval, error_code& ec) noexcept;
```

The function returns information about the volume chosen by *pval*, in a structure of type `space_info`. The structure contains `uintmax_t(-1)` for any value that can't be determined.

status

```
C++
```

```
file_status status(const path& pval);
file_status status(const path& pval, error_code& ec) noexcept;
```

The functions return the pathname status, the file type, and permissions, associated with *pval*. A symlink is itself not tested, but the file it chooses.

status_known

```
C++
```

```
bool status_known(file_status stat) noexcept;
```

The function returns `stat.type() != file_type::none`

swap

```
C++
```

```
void swap(path& left, path& right) noexcept;
```

The function exchanges the contents of *left* and *right*.

symlink_status

```
```

```
C++

file_status symlink_status(const path& pval);
file_status symlink_status(const path& pval, error_code& ec) noexcept;
```

The functions return the pathname symlink status, the file type, and permissions, associated with *pval*. The functions behave the same as `status(pval)` except that a symlink is itself tested, not the file it chooses.

system_complete

```
C++

path system_complete(const path& pval);
path system_complete(const path& pval, error_code& ec);
```

The functions return an absolute pathname that takes into account, as necessary, the current directory associated with its root name. (For POSIX, the functions return `absolute(pval)`).

temp_directory_path

```
C++

path temp_directory_path();
path temp_directory_path(error_code& ec);
```

The functions return a pathname for a directory suitable for containing temporary files.

u8path

```
C++

template <class Source>
path u8path(const Source& source);

template <class InIt>
path u8path(InIt first, InIt last);
```

The first function behaves the same as `path(source)` and the second function behaves the same as `path(first, last)` except that the chosen source in each case is taken as a

sequence of char elements encoded as UTF-8, whatever the filesystem.

`weakly_canonical`

C++

```
path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);
```

`<filesystem>` enumerations

Article • 12/07/2021

This topic documents the enums in the filesystem header.

Requirements

Header: <experimental/filesystem>

Namespace: std::experimental::filesystem

copy_options

An enumeration of bitmask values that is used with copy and copy_file functions to specify behavior.

Syntax

C++

```
enum class copy_options {
   none = 0,
   skip_existing = 1,
   overwrite_existing = 2,
   update_existing = 4,
   recursive = 8,
   copy_symlinks = 16,
   skip_symlinks = 32,
   directories_only = 64,
   create_symlinks = 128,
   create_hard_links = 256
};
```

Values

Name	Description
`none`	Perform the default behavior for the operation.
`skip_existing`	Do not copy if the file already exists, do not report an error.
`overwrite_existing`	Overwrite the file if it already exists.

Name	Description
`update_existing`	Overwrite the file if it already exists and is older than the replacement.
`recursive`	Recursively copy subdirectories and their contents.
`copy_symlinks`	Copy symbolic links as symbolic links, instead of copying the files they point to.
`skip_symlinks`	Ignore symbolic links.
`directories_only`	Only iterate over directories, ignore files.
`create_symlinks`	Make symbolic links instead of copying files. An absolute path must be used as the source path unless the destination is the current directory.
`create_hard_links`	Make hard links instead of copying files.

directory_options

Specifies whether to follow symbolic links to directories or to ignore them.

Syntax

C++

```
enum class directory_options {
   none = 0,
   follow_directory_symlink
};
```

Values

Name	Description
`none`	Default behavior: ignore symbolic links to directories. Permission denied is an error.
`follow_directory_symlink`	Treat symbolic links to directories as actual directories.

file_type

An enumeration for file types. The supported values are regular, directory, not_found, and unknown.

Syntax

```
C++
```

```
enum class file_type {
    not_found = -1,
    none,
    regular,
    directory,
    symlink,
    block,
    character,
    fifo,
    socket,
    unknown
};
```

Values

Name	Value	Description
`not_found`	-1	Represents a file that does not exist.
`none`	0	Represents a file that has no type attribute. (Not supported.)
`regular`	1	Represents a conventional disk file.
`directory`	2	Represents a directory.
`symlink`	3	Represents a symbolic link. (Not supported.)
`block`	4	Represents a block-special file on UNIX-based systems. (Not supported.)
`character`	5	Represents a character-special file on UNIX-based systems. (Not supported.)
`fifo`	6	Represents a FIFO file on UNIX-based systems. (Not supported.)
`socket`	7	Represents a socket on UNIX based systems. (Not supported.)
`unknown`	8	Represents a file whose status cannot be determined.

perm_options

Includes values `replace`, `add`, `remove`, and `nofollow`.

```
C++
```

```
enum class perm_options;
```

perms

Flags for file permissions. The supported values are essentially "readonly" and all. For a readonly file, none of the *_write bits are set. Otherwise the `all` bit (0x0777) is set.

Syntax

C++

```
enum class perms {// names for permissions
    none = 0,
    owner_read = 0400,  // S_IRUSR
    owner_write = 0200, // S_IWUSR
    owner_exec = 0100,  // S_IXUSR
    owner_all = 0700,   // S_IRWXU
    group_read = 040,   // S_IRGRP
    group_write = 020,  // S_IWGRP
    group_exec = 010,   // S_IXGRP
    group_all = 070,    // S_IRWXG
    others_read = 04,   // S_IROTH
    others_write = 02,  // S_IWOTH
    others_exec = 01,   // S_IXOTH
    others_all = 07,    // S_IRWXO
    all = 0777,
    set_uid = 04000,    // S_ISUID
    set_gid = 02000,    // S_ISGID
    sticky_bit = 01000, // S_ISVTX
    mask = 07777,
    unknown = 0xFFFF,
    add_perms = 0x10000,
    remove_perms = 0x20000,
    resolve_symlinks = 0x40000
};
```

See also

Header Files Reference
<filesystem>

directory_entry Class

Article • 06/16/2022

Describes an object that is returned by `*X`, where *X* is a directory_iterator or a recursive_directory_iterator.

Syntax

```
C++

class directory_entry;
```

Remarks

The class stores an object of type path. The stored `path` can be an instance of the path Class or of a type that is derived from `path`. It also stores two file_type values. One value represents the status of the stored file name. The other represents the symbolic link status of the file name.

For more information and code examples, see File System Navigation (C++).

Constructors

Constructor	Description
directory_entry	The defaulted constructors behave as expected. The fourth constructor initializes `mypath` to *pval*, `mystat` to *stat_arg*, and `mysymstat` to *symstat_arg*.

Member functions

Member function	Description
assign	The member function assigns *pval* to `mypath`, *stat* to `mystat`, and *symstat* to `mysymstat`.
path	The member function returns `mypath`.
replace_filename	The member function replaces `mypath` with `mypath.parent_path()` / *pval*, `mystat` with *stat_arg*, and `mysymstat` with *symstat_arg*

Member function	Description
status	Both member functions return `mystat` possibly first altered.
symlink_status	Both member functions return `mysymstat` possibly first altered.

Operators

Operator	Description
operator!=	Replaces the elements of the list with a copy of another list.
operator=	The defaulted member assignment operators behave as expected.
operator==	Returns `mypath == right.mypath`.
operator<	Returns `mypath < right.mypath`.
operator<=	Returns `!(right < *this)`.
operator>	Returns `right < *this`.
operator>=	Returns `!(*this < right)`.
operator const path_type&	Returns `mypath`.

Requirements

Header: <experimental/filesystem>

Namespace: std::experimental::filesystem

assign

The member function assigns *pval* to `mypath`, *stat_arg* to `mystat`, and *symstat_arg* to `mysymstat`.

```
C++
```

```
void assign(const std::experimental::filesystem::path& pval,
    file_status stat_arg = file_status(),
    file_status symstat_arg = file_status());
```

Parameters

pval
The stored file name path.

stat_arg
The status of the stored file name.

symstat_arg
The symbolic link status of the stored file name.

directory_entry

The defaulted constructors behave as expected. The fourth constructor initializes `mypath` to *pval*, `mystat` to *stat_arg*, and `mysymstat` to *symstat_arg*.

```
C++
```

```
directory_entry() = default;
directory_entry(const directory_entry&) = default;
directory_entry(directory_entry&&) noexcept = default;
explicit directory_entry(const std::experimental::filesystem::path& pval,
    file_status stat_arg = file_status(),
    file_status symstat_arg = file_status());
```

Parameters

pval
The stored file name path.

stat_arg
The status of the stored file name.

symstat_arg
The symbolic link status of the stored file name.

operator!=

The member function returns `!(*this == right)`.

```
C++
```

```
bool operator!=(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

operator=

The defaulted member assignment operators behave as expected.

```
C++
```

```
directory_entry& operator=(const directory_entry&) = default;
directory_entry& operator=(directory_entry&&) noexcept = default;
```

Parameters

right
The directory_entry being copied into the `directory_entry`.

operator==

The member function returns `mypath == right.mypath`.

```
C++
```

```
bool operator==(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

`operator<`

The member function returns `mypath < right.mypath`.

```
C++
```

```
bool operator<(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

`operator<=`

The member function returns `!(right < *this)`.

```
C++
```

```
bool operator<=(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

`operator>`

The member function returns `right < *this`.

```
C++
```

```
bool operator>(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

`operator>=`

The member function returns `!(*this < right)`.

```
C++
```

```
bool operator>=(const directory_entry& right) const noexcept;
```

Parameters

right
The directory_entry being compared to the `directory_entry`.

operator const path_type&

The member operator returns `mypath`.

```
C++
```

```
operator const std::experimental::filesystem::path&() const;
```

path

The member function returns `mypath`.

```
C++
```

```
const std::experimental::filesystem::path& path() const noexcept;
```

replace_filename

The member function replaces `mypath` with `mypath.parent_path()` / *pval*, `mystat` with *stat_arg*, and `mysymstat` with *symstat_arg*

```
C++
```

```
void replace_filename(
    const std::experimental::filesystem::path& pval,
    file_status stat_arg = file_status(),
    file_status symstat_arg = file_status());
```

Parameters

pval
The stored file name path.

stat_arg
The status of the stored file name.

symstat_arg
The symbolic link status of the stored file name.

status

Both member functions return `mystat` possibly first altered as follows:

1. If `status_known(mystat)` then do nothing.
2. Otherwise, if `!status_known(mysymstat) && !is_symlink(mysymstat)` then `mystat = mysymstat`.

```
C++
```

```
file_status status() const;
file_status status(error_code& ec) const noexcept;
```

Parameters

ec
The status error code.

symlink_status

Both member functions return `mysymstat` possibly first altered as follows: If `status_known(mysymstat)` then do nothing. Otherwise, `mysymstat = symlink_status(mypval)`.

```
C++
```

```
file_status symlink_status() const;
file_status symlink_status(error_code& ec) const noexcept;
```

Parameters

ec
The status error code.

See also

Header Files Reference

<filesystem>

`directory_iterator` class

Article • 05/01/2023

Describes an input iterator that sequences through the filenames in a directory. For an iterator `X`, the expression `*X` evaluates to an object of class `directory_entry` that wraps the filename and anything known about its status.

The class stores an object of type `path`, called `mydir` here for the purposes of exposition, which represents the name of the directory to be sequenced, and an object of type `directory_entry` called `myentry` here, which represents the current filename in the directory sequence. A default constructed object of type `directory_entry` has an empty `mydir` pathname and represents the end-of-sequence iterator.

For example, given the directory `abc` with entries `def` and `ghi`, the code:

`for (directory_iterator next(path("abc")), end; next != end; ++next) visit(next->path());`

calls `visit` with the arguments `path("abc/def")` and `path("abc/ghi")`.

For more information and code examples, see File System Navigation (C++).

Syntax

C++

```
class directory_iterator;
```

Constructors

Constructor	Description
directory_iterator	Constructs an input iterator that sequences through the filenames in a directory.

Member functions

Member function	Description
increment	Attempts to advance to the next filename in the directory.

Operators

Operator	Description
operator!=	Returns `!(*this == right)`.
operator=	The defaulted member assignment operators behave as expected.
operator==	Returns **`true`** only if both **`*this`** and *`right`* are end-of-sequence iterators or both aren't end-of-sequence-iterators.
operator*	Returns `myentry`.
operator->	Returns `&**this`.
operator++	Calls `increment()`, then returns **`*this`**, or makes a copy of the object, calls `increment()`, then returns the copy.

Requirements

Header: `<filesystem>`

Namespace: `std::filesystem`

`directory_iterator::directory_iterator`

The first constructor produces an end-of-sequence iterator. The second and third constructors store *`pval`* in `mydir`, then attempt to open and read `mydir` as a directory. If successful, they store the first filename in the directory in `myentry`; otherwise they produce an end-of-sequence iterator.

The default constructor behaves as expected.

C++

```
directory_iterator() noexcept;
explicit directory_iterator(const path& pval);

directory_iterator(const path& pval, error_code& ec) noexcept;
directory_iterator(const directory_iterator&) = default;
directory_iterator(directory_iterator&&) noexcept = default;
```

Parameters

pval

The stored file name path.

ec

The status error code.

directory_iterator

The stored object.

directory_iterator::increment

The function attempts to advance to the next filename in the directory. If successful, it stores that filename in `myentry`; otherwise it produces an end-of-sequence iterator.

```
C++

directory_iterator& increment(error_code& ec) noexcept;
```

directory_iterator::operator!=

The member operator returns `!(*this == right)`.

```
C++

bool operator!=(const directory_iterator& right) const;
```

Parameters

right

The directory_iterator being compared to the `directory_iterator`.

directory_iterator::operator=

The defaulted member assignment operators behave as expected.

```
C++

directory_iterator& operator=(const directory_iterator&) = default;
directory_iterator& operator=(directory_iterator&&) noexcept = default;
```

Parameters

right

The directory_iterator being copied into the `directory_iterator`.

directory_iterator::operator==

The member operator returns **`true`** only if both **`*this`** and *right* are end-of-sequence iterators or both aren't end-of-sequence-iterators.

```
C++
```

```
bool operator==(const directory_iterator& right) const;
```

Parameters

right

The directory_iterator being compared to the `directory_iterator`.

directory_iterator::operator*

The member operator returns `myentry`.

```
C++
```

```
const directory_entry& operator*() const;
```

directory_iterator::operator->

The member function returns `&**this`.

```
C++
```

```
const directory_entry * operator->() const;
```

directory_iterator::operator++

The first member function calls `increment()`, then returns `*this`. The second member function makes a copy of the object, calls `increment()`, then returns the copy.

```
C++

directory_iterator& operator++();
directory_iterator& operator++(int);
```

Parameters

`int`

The number of increments.

See also

<filesystem>
Header Files Reference
File System Navigation (C++)

file_status Class

Article • 08/03/2021

Wraps a file_type and file perms.

Syntax

```
C++

class file_status;
```

Constructors

Constructor	Description
file_status	Constructs a wrapper for file_type and file perms.

Member functions

Member function	Description
type	Gets or sets the `file_type`.
permissions	Gets or sets the file permissions.

Operators

Operator	Description
operator=	The defaulted member assignment operators behave as expected.

Requirements

Header: <filesystem>

Namespace: std::experimental::filesystem, std::experimental::filesystem

file_status::file_status

Constructs a wrapper for file_type and file perms.

```
C++
```

```
explicit file_status(
   file_type ftype = file_type::none,
   perms mask = perms::unknown) noexcept;

file_status(const file_status&) noexcept = default;

file_status(file_status&&) noexcept = default;

~file_status() noexcept = default;
```

Parameters

ftype
Specified `file_type`, defaults to `file_type::none`.

mask
Specified file `perms`, defaults to `perms::unknown`.

file_status
The stored object.

file_status::operator=

The defaulted member assignment operators behave as expected.

```
C++
```

```
file_status& operator=(const file_status&) noexcept = default;
file_status& operator=(file_status&&) nexcept = default;
```

Parameters

file_status
The file_status being copied into the `file_status`.

type

Gets or sets the `file_type`.

C++

```
file_type type() const noexcept
void type(file_type ftype) noexcept
```

Parameters

ftype
Specified `file_type`.

permissions

Gets or sets the file permissions.

Use the setter to make a file `readonly` or remove the `readonly` attribute.

C++

```
perms permissions() const noexcept
void permissions(perms mask) noexcept
```

Parameters

mask
Specified `perms`.

See also

Header Files Reference
path Class
<filesystem>

filesystem_error Class

Article • 08/03/2021

A base class for all exceptions that are thrown to report a low-level system overflow.

Syntax

```
C++
```

```
class filesystem_error    : public system_error;
```

Remarks

The class serves as the base class for all exceptions thrown to report an error in <filesystem> functions. It stores an object of type `string`, called `mymesg` here for the purposes of exposition. It also stores two objects of type `path`, called `mypval1` and `mypval2`.

Members

Constructors

Name	Description
filesystem_error	Constructs a `filesystem_error` message.

Functions

Name	Description
path1	Returns `mypval1`
path2	Returns `mypval2`
what	Returns a pointer to an `NTBS`.

Requirements

Header: <filesystem>

Namespace: std::experimental::filesystem

filesystem_error

The first constructor constructs its message from *what_arg* and *ec*. The second constructor also constructs its message from *pval1*, which it stores in `mypval1`. The third constructor also constructs its message from *pval1*, which it stores in `mypval1`, and from *pval2*, which it stores in `mypval2`.

C++

```
filesystem_error(const string& what_arg,
    error_code ec);

filesystem_error(const string& what_arg,
    const path& pval1,
    error_code ec);

filesystem_error(const string& what_arg,
    const path& pval1,
    const path& pval2,
    error_code ec);
```

Parameters

what_arg
Specified message.

ec
Specified error code.

mypval1
Further specified message parameter.

mypval2
Further specified message parameter.

path1

The member function returns `mypval1`

C++

```
const path& path1() const noexcept;
```

path2

The member function returns `mypval2`

```
C++
```

```
const path& path2() const noexcept;
```

what

The member function returns a pointer to an `NTBS`, preferably composed from `runtime_error::what()`, `system_error::what()`, `mymesg`, `mypval1.native_string()`, and `mypval2.native_string()`.

```
C++
```

```
const char *what() const noexcept;
```

`path` Class

Article • 11/04/2022

The `path` class stores an object of type `string_type`, called `myname` here for the purposes of exposition, suitable for use as a pathname. `string_type` is a synonym for `basic_string<value_type>`, where `value_type` is a synonym for `wchar_t` on Windows or `char` on POSIX.

For more information, and code examples, see File System Navigation (C++).

Syntax

```
C++

class path;
```

Constructors

Constructor	Description
path	Constructs a `path`.

Typedefs

Type name	Description
const_iterator	A synonym for `iterator`.
iterator	A bidirectional constant iterator that designates the `path` components of `myname`.
string_type	The type is a synonym for `basic_string<value_type>`.

Member functions

Member function	Description
append	Appends the specified sequence to `mypath`, converted and inserting a preferred_separator as needed.
assign	Replaces `mypath` with the specified sequence, converted as needed.

Member function	Description
begin	Returns a `path::iterator` designating the first path element in the pathname, if present.
c_str	Returns a pointer to the first character in `mypath`.
clear	Executes `mypath.clear()`.
compare	Returns comparison values.
concat	Appends the specified sequence to `mypath`, converted (but not inserting a separator) as needed.
empty	Returns `mypath.empty()`.
end	Returns an end-of-sequence iterator of type `iterator`.
extension	Returns the suffix of `filename()`.
filename	Returns the root directory component of `myname`, specifically `empty() ? path() : *--end()`. The component may be empty.
generic_string	Returns `this->string<Elem, Traits, Alloc>(al)` with (under Windows) any backslash converted to a forward slash.
generic_u16string	Returns `u16string()` with (under Windows) any backslash converted to a forward slash.
generic_u32string	Returns `u32string()` with (under Windows) any backslash converted to a forward slash.
generic_u8string	Returns `u8string()` with (under Windows) any backslash converted to a forward slash.
generic_wstring	Returns `wstring()` with (under Windows) any backslash converted to a forward slash.
has_extension	Returns `!extension().empty()`.
has_filename	Returns `!filename().empty()`.
has_parent_path	Returns `!parent_path().empty()`.
has_relative_path	Returns `!relative_path().empty()`.
has_root_directory	Returns `!root_directory().empty()`.
has_root_name	Returns `!root_name().empty()`.
has_root_path	Returns `!root_path().empty()`.

Member function	Description
has_stem	Returns `!stem().empty()`.
is_absolute	For Windows, the function returns `has_root_name() && has_root_directory()`. For POSIX, the function returns `has_root_directory()`.
is_relative	Returns `!is_absolute()`.
make_preferred	Converts each separator to a `preferred_separator` as needed.
native	Returns the native representation of the path.
parent_path	Returns the parent path component of `myname`.
preferred_separator	The constant object gives the preferred character for separating path components, depending on the host operating system.
relative_path	Returns the relative path component of `myname`.
remove_filename	Removes the filename.
replace_extension	Replaces the extension of `myname`.
replace_filename	Replaces the filename.
root_directory	Returns the root directory component of `myname`.
root_name	Returns the root name component of `myname`.
root_path	Returns the root path component of `myname`.
stem	Returns the `stem` component of `myname`.
string	Converts the sequence stored in `mypath`.
swap	Executes `swap(mypath, right.mypath)`.
u16string	Converts the sequence stored in `mypath` to UTF-16 and returns it stored in an object of type `u16string`.
u32string	Converts the sequence stored in `mypath` to UTF-32 and returns it stored in an object of type `u32string`.
u8string	Converts the sequence stored in `mypath` to UTF-8 and returns it stored in an object of type `u8string`.
value_type	The type describes the path elements favored by the host operating system.

Member function	Description
wstring	Converts the sequence stored in `mypath` to the encoding favored by the host system for a `wchar_t` sequence and returns it stored in an object of type `wstring`.

Operators

Operator	Description
operator=	Replaces the elements of the path with a copy of another path.
operator+=	Various `concat` expressions.
operator/=	Various `append` expressions.
operator string_type	Returns `myname`.

Requirements

Header: `<filesystem>`

Namespace: `std::experimental::filesystem`

`path::append`

Appends the specified sequence to `mypath`, converted and inserting a `preferred_separator` as needed.

```
C++
```

```
template <class Source>
path& append(const Source& source);

template <class InIt>
path& append(InIt first, InIt last);
```

Parameters

`source`

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

path::assign

Replaces `mypath` with the specified sequence, converted as needed.

```
C++
```

```
template <class Source>
path& assign(const Source& source);

template <class InIt>
path& assign(InIt first, InIt last);
```

Parameters

source

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

path::begin

Returns a `path::iterator` designating the first path element in the pathname, if present.

```
C++
```

```
iterator begin() const;
```

path::c_str

Returns a pointer to the first character in `mypath`.

```
```

C++

```
const value_type& *c_str() const noexcept;
```

`path::clear`

Executes `mypath.clear()`.

C++

```
void clear() noexcept;
```

`path::compare`

The first function returns `mypath.compare(pval.native())`. The second function returns `mypath.compare(str)`. The third function returns `mypath.compare(ptr)`.

C++

```
int compare(const path& pval) const noexcept;
int compare(const string_type& str) const;
int compare(const value_type *ptr) const;
```

Parameters

`pval`
Path to compare.

`str`
String to compare.

`ptr`
Pointer to compare.

`path::concat`

Appends the specified sequence to `mypath`, converted (but not inserting a separator) as needed.

C++

```
template <class Source>
path& concat(const Source& source);

template <class InIt>
path& concat(InIt first, InIt last);
```

Parameters

source

Specified sequence.

first

Start of specified sequence.

last

End of specified sequence.

path::const_iterator

A synonym for `iterator`.

C++

```
typedef iterator const_iterator;
```

path::empty

Returns `mypath.empty()`.

C++

```
bool empty() const noexcept;
```

path::end

Returns an end-of-sequence iterator of type `iterator`.

C++

```
iterator end() const;
```

`path::extension`

Returns the suffix of `filename()`.

```
C++

path extension() const;
```

Remarks

Returns the suffix of `filename() X` such that:

If `X == path(".") || X == path("..")` or if `X` contains no dot, the suffix is empty.

Otherwise, the suffix begins with (and includes) the rightmost dot.

`path::filename`

Returns the root directory component of `myname`, specifically `empty() path() : *--end()`. The component may be empty.

```
C++

path filename() const;
```

`path::generic_string`

Returns `this->string<Elem, Traits, Alloc>(al)` with (under Windows) any backslash converted to a forward slash.

```
C++

template <class Elem,
    class Traits = char_traits<Elem>,
    class Alloc = allocator<Elem>>
  basic_string<Elem, Traits, Alloc>
    generic_string(const Alloc& al = Alloc()) const;

string generic_string() const;
```

`path::generic_u16string`

Returns `u16string()` with (under Windows) any backslash converted to a forward slash.

```
C++

u16string generic_u16string() const;
```

`path::generic_u32string`

Returns `u32string()` with (under Windows) any backslash converted to a forward slash.

```
C++

u32string generic_u32string() const;
```

`path::generic_u8string`

Returns `u8string()` with (under Windows) any backslash converted to a forward slash.

```
C++

string generic_u8string() const;
```

`path::generic_wstring`

Returns `wstring()` with (under Windows) any backslash converted to a forward slash.

```
C++

wstring generic_wstring() const;
```

`path::has_extension`

Returns `!extension().empty()`.

```
C++

bool has_extension() const;
```

`path::has_filename`

Returns `!filename().empty()`.

```
C++

bool has_filename() const;
```

`path::has_parent_path`

Returns `!parent_path().empty()`.

```
C++

bool has_parent_path() const;
```

`path::has_relative_path`

Returns `!relative_path().empty()`.

```
C++

bool has_relative_path() const;
```

`path::has_root_directory`

Returns `!root_directory().empty()`.

```
C++

bool has_root_directory() const;
```

`path::has_root_name`

Returns `!root_name().empty()`.

```
C++

bool has_root_name() const;
```

`path::has_root_path`

Returns `!root_path().empty()`.

```
C++

bool has_root_path() const;
```

`path::has_stem`

Returns `!stem().empty()`.

```
C++

bool has_stem() const;
```

`path::is_absolute`

For Windows, the function returns `has_root_name() && has_root_directory()`. For POSIX, the function returns `has_root_directory()`.

```
C++

bool is_absolute() const;
```

`path::is_relative`

Returns `!is_absolute()`.

```
C++

bool is_relative() const;
```

`path::iterator`

A bidirectional constant iterator that designates the path components of `myname`.

```
C++
```

```
class iterator
   {
   // bidirectional iterator for path
   typedef bidirectional_iterator_tag iterator_category;
   typedef path_type value_type;
   typedef ptrdiff_t difference_type;
   typedef const value_type *pointer;
   typedef const value_type& reference;
   // ...
   };
```

Remarks

The class describes a bidirectional constant iterator that designates the `path` components of `myname` in the sequence:

1. the root name, if present
2. the root directory, if present
3. the remaining directory elements of the parent `path`, if present, ending with the filename, if present

For `pval` an object of type `path`:

1. `path::iterator X = pval.begin()` designates the first `path` element in the pathname, if present.
2. `X == pval.end()` is `true` when `X` points just past the end of the sequence of components.
3. `*X` returns a string that matches the current component
4. `++X` designates the next component in the sequence, if present.
5. `--X` designates the preceding component in the sequence, if present.
6. Altering `myname` invalidates all iterators designating elements in `myname`.

`path::make_preferred`

Converts each separator to a `preferred_separator` as needed.

```
C++
```

```
path& make_preferred();
```

path::native

Get the native string representation of the path.

C++

```
const string_type& native() const noexcept;
```

Remarks

The path is available in a portable generic format (see generic_string()) or the native format of the path. This function returns the native string. On a POSIX system, the generic format and the native format are the same.

In the following example running on Windows 11, the generic path string is `c:/t/temp/temp.txt` and the native string is `c:\\t\\temp.txt`

C++

```
// Compile with /std:c++17 or higher
#include <filesystem>

int main()
{
    std::filesystem::path p(R"(c:\t\temp.txt)");
    auto native = p.native(); // Windows: L"c:\\t\temp.txt"
    auto generic = p.generic_string(); // Windows: "c:/t/temp.txt"
}
```

path::operator=

Replaces the elements of the path with a copy of another path.

C++

```
path& operator=(const path& right);
path& operator=(path&& right) noexcept;

template <class Source>
path& operator=(const Source& source);
```

Parameters

right

The path being copied into the `path`.

source

The source `path`.

Remarks

The first member operator copies `right.myname` to `myname`. The second member operator moves `right.myname` to `myname`. The third member operator behaves the same as `*this = path(source)`.

`path::operator+=`

Various `concat` expressions.

C++

```
path& operator+=(const path& right);
path& operator+=(const string_type& str);
path& operator+=(const value_type *ptr);
path& operator+=(value_type elem);

template <class Source>
path& operator+=(const Source& source);

template <class Elem>
path& operator+=(Elem elem);
```

Parameters

right

The added path.

str

The added string.

ptr

The added pointer.

elem

The added `value_type` or `Elem`.

source

The added source.

Remarks

The member functions behave the same as the following corresponding expressions:

1. `concat(right);`
2. `concat(path(str));`
3. `concat(ptr);`
4. `concat(string_type(1, elem));`
5. `concat(source);`
6. `concat(path(basic_string<Elem>(1, elem)));`

`path::operator/=`

Various `append` expressions.

```
C++
```

```
path& operator/=(const path& right);

template <class Source>
path& operator/=(const Source& source);
```

Parameters

right

The added path.

source

The added source.

Remarks

The member functions behave the same as the following corresponding expressions:

1. `append(right);`
2. `append(source);`

`path::operator string_type`

Returns `myname`.

```
C++

operator string_type() const;
```

`path::parent_path`

Returns the parent path component of `myname`.

```
C++

path parent_path() const;
```

Remarks

Returns the parent path component of `myname`, specifically the prefix of `myname` after removing `filename().native()` and any immediately preceding directory separators. (Equally, if `begin() != end()`, it's the combining of all elements in the range `[begin(), --end())` by successively applying `operator/=`.) The component may be empty.

`path::path`

Constructs a `path` in various ways.

```
C++

path();

path(const path& right);
path(path&& right) noexcept;

template <class Source>
path(const Source& source);
```

```
template <class Source>
path(const Source& source, const locale& loc);

template <class InIt>
path(InIt first, InIt last);

template <class InIt>
path(InIt first, InIt last, const locale& loc);
```

Parameters

right

The path of which the constructed path is to be a copy.

source

The source of which the constructed path is to be a copy.

loc

The specified locale.

first

The position of the first element to be copied.

last

The position of the last element to be copied.

Remarks

The constructors all construct `myname` in various ways:

For `path()` it's `myname()`.

For `path(const path& right)` it's `myname(right.myname)`.

For `path(path&& right)` it's `myname(right.myname)`.

For `template<class Source> path(const Source& source)` it's `myname(source)`.

For `template<class Source> path(const Source& source, const locale& loc)` it's `myname(source)`, obtaining any needed `codecvt` facets from `loc`.

For `template<class InIt> path(InIt first, InIt last)` it's `myname(first, last)`.

For `template<class InIt> path(InIt first, InIt last, const locale& loc)` it's `myname(first, last)`, obtaining any needed `codecvt` facets from `loc`.

`path::preferred_separator`

The constant object gives the preferred character for separating path components, depending on the host operating system.

C++

```
#if _WIN32_C_LIB
static constexpr value_type preferred_separator == L'\\';
#else // assume POSIX
static constexpr value_type preferred_separator == '/';
#endif // filesystem model now defined
```

Remarks

It's equally permissible in most contexts under Windows to use `L'/'` in its place.

`path::relative_path`

Returns the relative path component of `myname`.

C++

```
path relative_path() const;
```

Remarks

Returns the relative path component of `myname`, specifically the suffix of `myname` after removing `root_path().native()` and any immediately subsequent redundant directory separators. The component may be empty.

`path::remove_filename`

Removes the filename.

C++

```
path& remove_filename();
```

`path::replace_extension`

Replaces the extension of `myname`.

```
C++
```

```
path& replace_extension(const path& newext = path());
```

Parameters

`newext`
The new extension.

Remarks

First removes the suffix `extension().native()` from `myname`. Then if `!newext.empty() && newext[0] != dot` (where *dot* is `*path(".").c_str()`), then *dot* is appended to `myname`. Then *`newext`* is appended to `myname`.

`path::replace_filename`

Replaces the filename.

```
C++
```

```
path& replace_filename(const path& pval);
```

Parameters

`pval`
The path of the filename.

Remarks

The member function executes:

```
C++
```

```
remove_filename();

*this /= pval;
return (*this);
```

`path::root_directory`

Returns the root directory component of `myname`.

```
C++

path root_directory() const;
```

Remarks

The component may be empty.

`path::root_name`

Returns the root name component of `myname`.

```
C++

path root_name() const;
```

Remarks

The component may be empty.

`path::root_path`

Returns the root path component of `myname`.

```
C++

path root_path() const;
```

Remarks

Returns the root path component of `myname`, specifically `root_name()` / `root_directory`. The component may be empty.

`path::stem`

Returns the `stem` component of `myname`.

```
C++

path stem() const;
```

Remarks

Returns the `stem` component of `myname`, specifically `filename().native()` with any trailing `extension().native()` removed. The component may be empty.

`path::string`

Converts the sequence stored in `mypath`.

```
C++

template \<class Elem, class Traits = char_traits\<Elem>, class Alloc =
allocator\<Elem>>
basic_string\<Elem, Traits, Alloc> string(const Alloc& al = Alloc()) const;
string string() const;
```

Remarks

The first (template) member function converts the sequence stored in `mypath` the same way as:

1. `string()` for `string<char, Traits, Alloc>()`
2. `wstring()` for `string<wchar_t, Traits, Alloc>()`
3. `u16string()` for `string<char16_t, Traits, Alloc>()`
4. `u32string()` for `string<char32_t, Traits, Alloc>()`

The second member function converts the sequence stored in `mypath` to the encoding favored by the host system for a `char` sequence and returns it stored in an object of type `string`.

`path::string_type`

The type is a synonym for `basic_string<value_type>`.

```
C++

typedef basic_string<value_type> string_type;
```

`path::swap`

Executes `swap(mypath, right.mypath)`.

```
C++

void swap(path& right) noexcept;
```

`path::u16string`

Converts the sequence stored in `mypath` to UTF-16 and returns it stored in an object of type `u16string`.

```
C++

u16string u16string() const;
```

`path::u32string`

Converts the sequence stored in `mypath` to UTF-32 and returns it stored in an object of type `u32string`.

```
C++

u32string u32string() const;
```

`path::u8string`

Converts the sequence stored in `mypath` to UTF-8 and returns it stored in an object of type `u8string`.

```
C++
```

```
string u8string() const;
```

`path::value_type`

The type describes the `path` elements favored by the host operating system.

```
C++
```

```
#if _WIN32_C_LIB
typedef wchar_t value_type;
#else // assume POSIX
typedef char value_type;
#endif // filesystem model now defined
```

`path::wstring`

Converts the sequence stored in `mypath` to the encoding favored by the host system for a `wchar_t` sequence and returns it stored in an object of type `wstring`.

```
C++
```

```
wstring wstring() const;
```

See also

Header Files Reference

recursive_directory_iterator Class

Article • 06/20/2022

Describes an input iterator that sequences through the filenames in a directory, possibly descending into subdirectories recursively. For an iterator `x`, the expression `*x` evaluates to an object of class `directory_entry` that wraps the filename and anything known about its status.

For more information and code examples, see File System Navigation (C++).

Syntax

```
C++
```

```
class recursive_directory_iterator;
```

Remarks

The class template stores:

1. an object of type `stack<pair<directory_iterator, path>>`, called `mystack` here for the purposes of exposition, which represents the nest of directories to be sequenced
2. an object of type `directory_entry` called `myentry` here, which represents the current filename in the directory sequence
3. an object of type `bool`, called `no_push` here, which records whether recursive descent into subdirectories is disabled
4. an object of type `directory_options`, called `myoptions` here, which records the options established at construction

A default constructed object of type `recursive_directory_entry` has an end-of-sequence iterator at `mystack.top().first` and represents the end-of-sequence iterator. For example, given the directory `abc` with entries `def` (a directory), `def/ghi`, and `jkl`, the code:

```
C++
```

```
for (recursive_directory_iterator next(path("abc")), end; next != end; ++next)
    visit(next->path());
```

will call visit with the arguments `path("abc/def/ghi")` and `path("abc/jkl")`. You can qualify sequencing through a directory subtree in two ways:

1. A directory symlink will be scanned only if you construct a `recursive_directory_iterator` with a `directory_options` argument whose value is `directory_options::follow_directory_symlink`.

2. If you call `disable_recursion_pending`, a subsequent directory encountered during an increment won't be recursively scanned.

Constructors

Constructor	Description
recursive_directory_iterator	Constructs a `recursive_directory_iterator`.

Member functions

Member function	Description
depth	Returns `mystack.size() - 1`, so `pval` is at depth zero.
disable_recursion_pending	Stores `true` in `no_push`.
increment	Advances to the next filename in sequence.
options	Returns `myoptions`.
pop	Returns the next object.
recursion_pending	Returns `!no_push`.

Operators

Operator	Description
operator!=	Returns `!(*this == right)`.
operator=	The defaulted member assignment operators behave as expected.
operator==	Returns `true` only if both `*this` and *right* are end-of-sequence iterators or both aren't end-of-sequence-iterators.
operator*	Returns `myentry`.
operator->	Returns `&**this`.
operator++	Increments the `recursive_directory_iterator`.

Requirements

Header: <filesystem>

Namespace: std::tr2::sys

recursive_directory_iterator::depth

Returns `mystack.size() - 1`, so `pval` is at depth zero.

```
C++

int depth() const;
```

recursive_directory_iterator::disable_recursion_pending

Stores **`true`** in `no_push`.

```
C++

void disable_recursion_pending();
```

recursive_directory_iterator::increment

Advances to the next filename in sequence.

```
C++

recursive_directory_iterator& increment(error_code& ec) noexcept;
```

Parameters

ec
Specified error code.

Remarks

The function attempts to advance to the next filename in the nested sequence. If successful, it stores that filename in `myentry`; otherwise it produces an end-of-sequence iterator.

recursive_directory_iterator::operator!=

Returns `!(*this == right)`.

```

```

```
C++

bool operator!=(const recursive_directory_iterator& right) const;
```

Parameters

right
The recursive_directory_iterator for comparison.

recursive_directory_iterator::operator=

The defaulted member assignment operators behave as expected.

```
C++

recursive_directory_iterator& operator=(const recursive_directory_iterator&) =
default;
recursive_directory_iterator& operator=(recursive_directory_iterator&&) noexcept =
default;
```

Parameters

recursive_directory_iterator
The recursive_directory_iterator being copied into the `recursive_directory_iterator`.

recursive_directory_iterator::operator==

Returns **`true`** only if both **`*this`** and *right* are end-of-sequence iterators or both aren't end-of-sequence-iterators.

```
C++

bool operator==(const recursive_directory_iterator& right) const;
```

Parameters

right
The recursive_directory_iterator for comparison.

recursive_directory_iterator::operator*

Returns `myentry`.

```
```

```
C++

const directory_entry& operator*() const;
```

recursive_directory_iterator::operator->

Returns `&**this`.

```
C++

const directory_entry * operator->() const;
```

recursive_directory_iterator::operator++

Increments the `recursive_directory_iterator`.

```
C++

recursive_directory_iterator& operator++();

recursive_directory_iterator& operator++(int);
```

Parameters

int
The specified increment.

Remarks

The first member function calls `increment()`, then returns `*this`. The second member function makes a copy of the object, calls `increment()`, then returns the copy.

recursive_directory_iterator::options

Returns `myoptions`.

```
C++

directory_options options() const;
```

recursive_directory_iterator::pop

Returns the next object.

```
C++
```

```
void pop();
```

Remarks

If `depth() == 0` the object becomes an end-of-sequence iterator. Otherwise, the member function terminates scanning of the current (deepest) directory and resumes at the next lower depth.

recursive_directory_iterator::recursion_pending

Returns `!no_push`.

```
C++
```

```
bool recursion_pending() const;
```

recursive_directory_iterator::recursive_directory_iterator

Constructs a `recursive_directory_iterator`.

```
C++
```

```
recursive_directory_iterator() noexcept;
explicit recursive_directory_iterator(const path& pval);

recursive_directory_iterator(const path& pval,
    error_code& ec) noexcept;
recursive_directory_iterator(const path& pval,
    directory_options opts);

recursive_directory_iterator(const path& pval,
    directory_options opts,
    error_code& ec) noexcept;
recursive_directory_iterator(const recursive_directory_iterator&) = default;
recursive_directory_iterator(recursive_directory_iterator&&) noexcept = default;
```

Parameters

pval
The specified path.

error_code
The specified error code.

opts
The specified directory options.

recursive_directory_iterator
The `recursive_directory_iterator` of which the constructed `recursive_directory_iterator` is to be a copy.

Remarks

The first constructor produces an end-of-sequence iterator. The second and third constructors store **`false`** in `no_push` and `directory_options::none` in `myoptions`, then attempt to open and read *pval* as a directory. If successful, they initialize `mystack` and `myentry` to designate the first non-directory filename in the nested sequence; otherwise they produce an end-of-sequence iterator.

The fourth and fifth constructors behave the same as the second and third, except that they first store *opts* in `myoptions`. The default constructor behaves as expected.

See also

Header Files Reference
<filesystem>
File System Navigation (C++)

space_info Structure

Article • 08/03/2021

Holds information about a volume.

Syntax

```
C++
```

```
struct space_info
{
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};
```

Members

Public Data Members

Name	Description
`unsigned long long capacity`	Represents the total number of bytes that the volume can represent.
`unsigned long long free`	Represents the number of bytes that are not used to represent data on the volume.
`unsigned long long available`	Represents the number of bytes that are available to represent data on the volume.

Requirements

Header: <filesystem>

Namespace: std::experimental::filesystem

See also

Header Files Reference

<filesystem>

File System Navigation (C++)

<forward_list>

Article • 12/07/2021

Defines the container class template forward_list and several supporting templates.

Requirements

Header: <forward_list>

Namespace: std

> ⓘ Note
>
> The <forward_list> library also uses the `#include <initializer_list>` statement.

Members

Operators

Name	Description
operator==	Tests if the forward list object on the left side of the operator is equal to the forward list object on the right side.
operator!=	Tests if the forward list object on the left side of the operator is not equal to the forward list object on the right side.
operator<	Tests if the forward list object on the left side of the operator is less than the forward list object on the right side.
operator<=	Tests if the forward list object on the left side of the operator is less than or equal to the forward list object on the right side.
operator>	Tests if the forward list object on the left side of the operator is greater than the forward list object on the right side.
operator>=	Tests if the forward list object on the left side of the operator is greater than or equal to the forward list object on the right side.

Functions

Name	Description
swap	Exchanges the elements of two forward lists.

Classes

Name	Description
forward_list	Describes an object that controls a varying-length sequence of elements. The sequence is stored as a singly-linked list of elements, each containing a member of type `Type`.

See also

Header Files Reference

`<forward_list>` functions

Article • 12/07/2021

swap

Exchanges the elements of two forward lists.

```
C++
```

```
void swap(forward_list <Type, Allocator>& left, forward_list <Type,
Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Remarks

This template function executes `left.swap(right)`.

`<forward_list>` operators

Article • 12/07/2021

operator==

Tests if the forward list object on the left side of the operator is equal to the forward list object on the right side.

```
C++
```

```
bool operator==(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Remarks

This template function overloads `operator==` to compare two objects of class template `forward_list`. The function returns `distance(left.begin(), end()) == distance(right.begin(),right.end()) && equal(left. begin(),left. end(),right.begin())`.

operator!=

Tests if the forward list object on the left side of the operator is not equal to the forward list object on the right side.

```
C++
```

```
bool operator!=(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Return Value

`true` if the lists are not equal; **`false`** if the lists are equal.

Remarks

This template function returns `!(left == right)`.

`operator<`

Tests if the forward list object on the left side of the operator is less than the forward list object on the right side.

```
C++
```

```
bool operator<(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Return Value

`true` if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise **`false`**.

Remarks

This template function overloads `operator<` to compare two objects of class template `forward_list`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

`operator<=`

Tests if the forward list object on the left side of the operator is less than or equal to the forward list object on the right side.

C++

```
bool operator<=(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Return Value

`true` if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise **`false`**.

Remarks

This template function returns `!(right < left)`.

`operator>`

Tests if the forward list object on the left side of the operator is greater than the forward list object on the right side.

C++

```
bool operator>(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Return Value

`true` if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **`false`**.

Remarks

This template function returns `right < left`.

`operator>=`

Tests if the forward list object on the left side of the operator is greater than or equal to the forward list object on the right side.

```
C++
```

```
bool operator>=(
    const forward_list <Type, Allocator>& left,
    const forward_list <Type, Allocator>& right);
```

Parameters

left
An object of type `forward_list`.

right
An object of type `forward_list`.

Return Value

`true` if the forward list on the left side of the operator is greater than or equal to the forward list on the right side of the operator; otherwise **`false`**.

Remarks

The template function returns `!(left < right)`.

forward_list Class

Article • 06/17/2022

Describes an object that controls a varying-length sequence of elements. The sequence is stored as a singly-linked list of nodes, each containing a member of type `Type`.

Syntax

C++

```
template <class Type,
    class Allocator = allocator<Type>>
class forward_list
```

Parameters

Type*
The element data type to be stored in the forward_list.

Allocator
The stored allocator object that encapsulates details about the forward_list allocation and deallocation of memory. This parameter is optional. The default value is allocator<`Type`>.

Remarks

A `forward_list` object allocates and frees storage for the sequence it controls through a stored object of class *Allocator* that is based on allocator Class (commonly known as `std::allocator`). For more information, see Allocators. An allocator object must have the same external interface as an object of type `allocator`.

> **Note**
>
> The stored allocator object is not copied when the container object is assigned.

Iterators, pointers and references might become invalid when elements of their controlled sequence are erased through `forward_list`. Insertions and splices performed on the controlled sequence through `forward_list` don't invalidate iterators.

Additions to the controlled sequence might occur by calls to forward_list::insert_after, which is the only member function that calls the constructor `Type(const T&)`. `forward_list` might also call move constructors. If such an expression throws an exception, the container object inserts no new elements and rethrows the exception. Thus, an object of type `forward_list` is left in a known state when such exceptions occur.

Members

Constructors

Name	Description
forward_list	Constructs an object of type `forward_list`.

Typedefs

Name	Description
allocator_type	A type that represents the allocator class for a forward list object.
const_iterator	A type that provides a constant iterator for the forward list.
const_pointer	A type that provides a pointer to a `const` element in a forward list.
const_reference	A type that provides a constant reference to an element in the forward list.
difference_type	A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.
iterator	A type that provides an iterator for the forward list.
pointer	A type that provides a pointer to an element in the forward list.
reference	A type that provides a reference to an element in the forward list.
size_type	A type that represents the unsigned distance between two elements.
value_type	A type that represents the type of element stored in a forward list.

Functions

Name	Description

Name	Description
assign	Erases elements from a forward list and copies a new set of elements to a target forward list.
before_begin	Returns an iterator addressing the position before the first element in a forward list.
begin	Returns an iterator addressing the first element in a forward list.
cbefore_begin	Returns a const iterator addressing the position before the first element in a forward list.
cbegin	Returns a const iterator addressing the first element in a forward list.
cend	Returns a const iterator that addresses the location succeeding the last element in a forward list.
clear	Erases all the elements of a forward list.
emplace_after	Move constructs a new element after a specified position.
emplace_front	Adds an element constructed in place to the beginning of the list.
empty	Tests whether a forward list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a forward list.
erase_after	Removes elements from the forward list after a specified position.
front	Returns a reference to the first element in a forward list.
get_allocator	Returns a copy of the allocator object used to construct a forward list.
insert_after	Adds elements to the forward list after a specified position.
max_size	Returns the maximum length of a forward list.
merge	Removes the elements from the argument list, inserts them into the target forward list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_front	Deletes the element at the beginning of a forward list.
push_front	Adds an element to the beginning of a forward list.
remove	Erases elements in a forward list that matches a specified value.
remove_if	Erases elements from a forward list for which a specified predicate is satisfied.
resize	Specifies a new size for a forward list.

Name	Description
reverse	Reverses the order in which the elements occur in a forward list.
sort	Arranges the elements in ascending order or with an order specified by a predicate.
splice_after	Restitches links between nodes.
swap	Exchanges the elements of two forward lists.
unique	Removes adjacent elements that pass a specified test.

Operators

Name	Description
operator=	Replaces the elements of the forward list with a copy of another forward list.

allocator_type

A type that represents the allocator class for a forward list object.

```
C++
```

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter Allocator.

assign

Erases elements from a forward list and copies a new set of elements to a target forward list.

```
C++
```

```
void assign(
    size_type Count,
    const Type& Val);

void assign(
    initializer_list<Type> IList);
```

```
template <class InputIterator>
void assign(InputIterator First, InputIterator Last);
```

Parameters

first
The beginning of the replacement range.

last
The end of the replacement range.

count
The number of elements to assign.

val
The value to assign each element.

Type
The type of the value.

IList
The initializer_list to copy.

Remarks

If the forward_list is an integer type, the first member function behaves the same as `assign((size_type)First, (Type)Last)`. Otherwise, the first member function replaces the sequence controlled by `*this` with the sequence [`First, Last)`, which must not overlap the initial controlled sequence.

The second member function replaces the sequence controlled by `*this` with a repetition of `Count` elements of value `Val`.

The third member function copies the elements of the initializer_list into the forward_list.

before_begin

Returns an iterator addressing the position before the first element in a forward list.

```
C++
```

```
const_iterator before_begin() const;
iterator before_begin();
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

Remarks

begin

Returns an iterator addressing the first element in a forward list.

C++

```
const_iterator begin() const;
iterator begin();
```

Return Value

A forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Remarks

cbefore_begin

Returns a const iterator addressing the position before the first element in a forward list.

C++

```
const_iterator cbefore_begin() const;
```

Return Value

A forward iterator that points just before the first element of the sequence (or just before the end of an empty sequence).

Remarks

cbegin

Returns a `const` iterator that addresses the first element in the range.

```
C++
```

```
const_iterator cbegin() const;
```

Return Value

A `const` forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++
```

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();
// i2 is Container<T>::const_iterator
```

cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container of any kind that supports `end()` and `cend()`.

```
C++

auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

clear

Erases all the elements of a forward list.

```
C++

void clear();
```

Remarks

This member function calls `erase_after(before_begin(), end()).`

const_iterator

A type that provides a constant iterator for the forward list.

```
C++
```

```
typedef implementation-defined const_iterator;
```

Remarks

`const_iterator` describes an object that can serve as a constant forward iterator for the controlled sequence. It's described here as a synonym for an implementation-defined type.

const_pointer

A type that provides a pointer to a **`const`** element in a forward list.

```
C++
```

```
typedef typename Allocator::const_pointer
    const_pointer;
```

Remarks

const_reference

A type that provides a constant reference to an element in the forward list.

```
C++
```

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

difference_type

A signed integer type that can be used to represent the number of elements of a forward list in a range between elements pointed to by iterators.

```
C++
```

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

`difference_type` describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

emplace_after

Move constructs a new element after a specified position.

```
C++
```

```
template <class T>
iterator emplace_after(const_iterator Where, Type&& val);
```

Parameters

Where
The position in the target forward list where the new element is constructed.

val
The constructor argument.

Return Value

An iterator that designates the newly inserted element.

Remarks

This member function inserts an element with the constructor arguments *val* just after the element pointed to by *Where* in the controlled sequence. Its behavior is otherwise the same as forward_list::insert_after.

emplace_front

Adds an element constructed in place to the beginning of the list.

```
C++
```

```
template <class Type>
    void emplace_front(Type&& val);
```

Parameters

val
The element added to the beginning of the forward list.

Remarks

This member function inserts an element with the constructor arguments `_ val` at the end of the controlled sequence.

If an exception is thrown, the container is left unaltered and the exception is rethrown.

empty

Tests whether a forward list is empty.

```
C++
```

```
bool empty() const;
```

Return Value

`true` if the forward list is empty; otherwise, **`false`**.

end

Returns an iterator that addresses the location succeeding the last element in a forward list.

```
C++
```

```
const_iterator end() const;
iterator end();
```

Return Value

A forward iterator that points just beyond the end of the sequence.

erase_after

Removes elements from the forward list after a specified position.

```
C++
```

```
iterator erase_after(const_iterator Where);
iterator erase_after(const_iterator first, const_iterator last);
```

Parameters

Where
The position in the target forward list where the element is erased.

first
The beginning of the range to erase.

last
The end of the range to erase.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or forward_list::end if no such element exists.

Remarks

The first member function removes the element of the controlled sequence just after *Where*.

The second member function removes the elements of the controlled sequence in the range `( first, last)` (neither end point is included).

Erasing `N` elements causes `N` destructor calls. Reallocation occurs, so iterators and references become invalid for the erased elements.

The member functions never throw an exception.

forward_list

Constructs an object of type `forward_list`.

```
C++
```

```
forward_list();
explicit forward_list(const Allocator& Al);
explicit forward_list(size_type Count);
forward_list(size_type Count, const Type& Val);
forward_list(size_type Count, const Type& Val, const Allocator& Al);
forward_list(const forward_list& Right);
forward_list(const forward_list& Right, const Allocator& Al);
forward_list(forward_list&& Right);
forward_list(forward_list&& Right, const Allocator& Al);
forward_list(initializer_list<Type> IList, const Alloc& Al);
template <class InputIterator>
forward_list(InputIterator First, InputIterator Last);
template <class InputIterator>
forward_list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

Al
The allocator class to use with this object.

Count
The number of elements in the list constructed.

Val
The value of the elements in the list constructed.

Right
The list of which the constructed list is to be a copy.

First
The position of the first element in the range of elements to be copied.

Last
The position of the first element beyond the range of elements to be copied.

IList
The initializer_list to copy.

Remarks

All constructors store an allocator and initialize the controlled sequence. The allocator object is the argument *Al*, if present. For the copy constructor, it's `right.get_allocator()`. Otherwise, it's `Allocator()`.

The first two constructors specify an empty initial controlled sequence.

The third constructor specifies a repetition of *Count* elements of value `Type()`.

The fourth and fifth constructors specify a repetition of *Count* elements of value *Val*.

The sixth constructor specifies a copy of the sequence controlled by *Right*. If `InputIterator` is an integer type, the next two constructors specify a repetition of `(size_type)First` elements of value `(Type)Last`. Otherwise, the next two constructors specify the sequence `[First, Last)`.

The ninth and tenth constructors are the same as the sixth, but with an rvalue reference.

The last constructor specifies the initial controlled sequence with an object of class `initializer_list<Type>`.

front

Returns a reference to the first element in a forward list.

C++

```
reference front();
const_reference front() const;
```

Return Value

A reference to the first element of the controlled sequence, which must be non-empty.

get_allocator

Returns a copy of the allocator object used to construct a forward list.

C++

```
allocator_type get_allocator() const;
```

Return Value

The stored allocator object.

insert_after

Adds elements to the forward list after a specified position.

```
C++
```

```
iterator insert_after(const_iterator Where, const Type& Val);
void insert_after(const_iterator Where, size_type Count, const Type& Val);
void insert_after(const iterator Where, initializer_list<Type> IList);
iterator insert_after(const_iterator Where, Type&& Val);
template <class InputIterator>
    void insert_after(const_iterator Where, InputIterator First,
InputIterator Last);
```

Parameters

Where
The position in the target forward list where the first element is inserted.

Count
The number of elements to insert.

First
The beginning of the insertion range.

Last
The end of the insertion range.

Val
The element added to the forward list.

IList
The initializer_list to insert.

Return Value

An iterator that designates the newly inserted element (first and last member functions only).

Remarks

Each of the member functions inserts—just after the element pointed to by *Where* in the controlled sequence—a sequence that' specified by the remaining operands.

The first member function inserts an element that has value *Val* and returns an iterator that designates the newly inserted element.

The second member function inserts a repetition of *Count* elements of value *Val*.

If `InputIterator` is an integer type, the third member function behaves the same as `insert(it, (size_type)First, (Type)Last)`. Otherwise, it inserts the sequence `[First, Last)`, which must not overlap the initial controlled sequence.

The fourth member function inserts the sequence that's specified by an object of class `initializer_list<Type>`.

The last member function is the same as the first, but with an rvalue reference.

Inserting `N` elements causes `N` constructor calls. Reallocation occurs, but no iterators or references become invalid.

If an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

iterator

A type that provides an iterator for the forward list.

```
C++

typedef implementation-defined iterator;
```

Remarks

`iterator` describes an object that can serve as a forward iterator for the controlled sequence. It's described here as a synonym for an implementation-defined type.

max_size

Returns the maximum length of a forward list.

```
C++

size_type max_size() const;
```

Return Value

The length of the longest sequence that the object can control.

Remarks

merge

Combines two sorted sequences into a single sorted sequence in linear time. Removes the elements from the argument list, and inserts them into this `forward_list`. The two lists should be sorted by the same compare function object before the call to `merge`. The combined list will be sorted by that compare function object.

C++

```
void merge(forward_list& right);
template <class Predicate>
    void merge(forward_list& right, Predicate comp);
```

Parameters

right
The forward list to merge from.

comp
The compare function object that is used to sort elements.

Remarks

`forward_list::merge` removes the elements from the `forward_list` `right`, and inserts them into this `forward_list`. Both sequences must be ordered by the same predicate, described below. The combined sequence is also ordered by that compare function object.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The second member function imposes the order `! comp(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `right`.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

operator=

Replaces the elements of the forward list with a copy of another forward list.

```
C++
```

```
forward_list& operator=(const forward_list& right);
forward_list& operator=(initializer_list<Type> IList);
forward_list& operator=(forward_list&& right);
```

Parameters

right
The forward list being copied into the forward list.

IList
A brace-enclosed initializer list, which behaves just like a sequence of elements of type `Type`.

Remarks

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *right*.

The second member operator replaces the controlled sequence from an object of class `initializer_list<Type>`.

The third member operator is the same as the first, but with an rvalue reference.

pointer

A type that provides a pointer to an element in the forward list.

```
C++
```

```
typedef typename Allocator::pointer pointer;
```

pop_front

Deletes the element at the beginning of a forward list.

```
C++
```

```
void pop_front();
```

Remarks

The first element of the forward list must be non-empty.

The member function never throws an exception.

push_front

Adds an element to the beginning of a forward list.

```
C++
```

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

val
The element added to the beginning of the forward list.

Remarks

If an exception is thrown, the container is left unaltered and the exception is rethrown.

reference

A type that provides a reference to an element in the forward list.

```
C++
```

```
typedef typename Allocator::reference reference;
```

remove

Erases elements in a forward list that matches a specified value.

```
C++
```

```
void remove(const Type& val);
```

Parameters

val
The value which, if held by an element, will result in that element's removal from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == val`.

The member function never throws an exception.

remove_if

Erases elements from a forward list for which a specified predicate is satisfied.

```
C++
```

```
template <class Predicate>
    void remove_if(Predicate pred);
```

Parameters

pred
The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Remarks

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `pred(*P)` is true.

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

resize

Specifies a new size for a forward list.

```
C++
```

```
void resize(size_type _Newsize);
void resize(size_type _Newsize, const Type& val);
```

Parameters

_Newsize
The number of elements in the resized forward list.

val
The value to use for padding.

Remarks

The member functions both ensure that the number of elements in the list henceforth is *_Newsize*. If it must make the controlled sequence longer, the first member function appends elements with value `Type()`, while the second member function appends elements with value *val*. To make the controlled sequence shorter, both member functions effectively call `erase_after(begin() + _Newsize - 1, end())`.

reverse

Reverses the order in which the elements occur in a forward list.

```
C++
```

```
void reverse();
```

size_type

A type that represents the unsigned distance between two elements.

```
```

```
C++
```

```
typedef typename Allocator::size_type size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence.

sort

Arranges the elements in ascending order or with an order specified by a predicate.

```
C++
```

```
void sort();
template <class Predicate>
void sort(Predicate pred);
```

Parameters

pred
The ordering predicate.

Remarks

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in `ascending` order.) The member template function imposes the order `!pred(*Pj, *Pi)` whenever `i < j`. No ordered pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. (The sort is stable.)

An exception occurs only if *pred* throws an exception. In that case, the controlled sequence is left in unspecified order and the exception is rethrown.

splice_after

Removes elements from a source forward_list and inserts them into a destination forward_list.

```
C++
```

```
// insert the entire source forward_list
void splice_after(const_iterator Where, forward_list& Source);
void splice_after(const_iterator Where, forward_list&& Source);

// insert one element of the source forward_list
void splice_after(const_iterator Where, forward_list& Source, const_iterator
Iter);
void splice_after(const_iterator Where, forward_list&& Source,
const_iterator Iter);

// insert a range of elements from the source forward_list
void splice_after(
    const_iterator Where,
    forward_list& Source,
    const_iterator First,
    const_iterator Last);

void splice_after(
    const_iterator Where,
    forward_list&& Source,
    const_iterator First,
    const_iterator Last);
```

Parameters

Where
The position in the destination forward_list after which to insert.

Source
The source forward_list that is to be inserted into the destination forward_list.

Iter
The element to be inserted from the source forward_list.

First
The first element in the range to be inserted from source forward_list.

Last
The first position beyond the range to be inserted from the source forward_list.

Remarks

The first pair of member functions inserts the sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes all elements from *Source*. (`&Source` must not equal `this`.)

The second pair of member functions removes the element just after *Iter* in the sequence controlled by *Source* and inserts it just after the element in the controlled sequence pointed to by *Where*. (If `Where == Iter || Where == ++Iter`, no change occurs.)

The third pair of member functions (ranged splice) inserts the subrange designated by `(First, Last)` from the sequence controlled by *Source* just after the element in the controlled sequence pointed to by *Where*. It also removes the original subrange from the sequence controlled by *Source*. (If `&Source == this`, the range `(First, Last)` must not include the element pointed to by *Where*.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class iterator is incremented `N` times.

No iterators, pointers, or references that designate spliced elements become invalid.

Example

C++

```
// forward_list_splice_after.cpp
// compile with: /EHsc /W4
#include <forward_list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    forward_list<int> c1{ 10, 11 };
    forward_list<int> c2{ 20, 21, 22 };
    forward_list<int> c3{ 30, 31 };
    forward_list<int> c4{ 40, 41, 42, 43 };

    forward_list<int>::iterator where_iter;
    forward_list<int>::iterator first_iter;
```

```
    forward_list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice_after(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);

    first_iter = c3.begin();
    c2.splice_after(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);

    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    c2.splice_after(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}
```

Output

```
Beginning state of lists:c1 = (10) (11)c2 = (20) (21) (22)c3 = (30) (31)c4 =
(40) (41) (42) (43)After splicing c1 into c2:c1 =c2 = (20) (21) (10) (11)
(22)After splicing the first element of c3 into c2:c3 = (30)c2 = (20) (21)
(31) (10) (11) (22)After splicing a range of c4 into c2:c4 = (40) (41)c2 =
(20) (21) (42) (43) (31) (10) (11) (22)
```

swap

Exchanges the elements of two forward lists.

```
C++
```

```
void swap(forward_list& right);
```

Parameters

right
The forward list providing the elements to be exchanged.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If `get_allocator() == right.get_allocator()`, it does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

unique

Eliminates all but the first element from every consecutive group of equal elements.

```
C++
```

```
void unique();
template <class BinaryPredicate>
void unique(BinaryPredicate comp);
```

Parameters

comp
The binary predicate used to compare successive elements.

Remarks

Keeps the first of each unique element, and removes the rest. The elements must be sorted so that elements of equal value are adjacent in the list.

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the second member function removes every element for which `i + 1 == j && comp(*Pi, *Pj)`.

For a controlled sequence of length `N` (> 0), the predicate `comp(*Pi, *Pj)` is evaluated `N - 1` times.

An exception occurs only if `comp` throws an exception. In that case, the controlled sequence is left in an unspecified state and the exception is rethrown.

value_type

A type that represents the type of element stored in a forward list.

```
C++

typedef typename Allocator::value_type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.

`<fstream>`

Article • 12/06/2021

Defines several classes that support iostreams operations on sequences stored in external files.

Syntax

```
C++
```

```
#include <fstream>
```

Typedefs

Type name	Description
filebuf	A type `basic_filebuf` specialized on `char` template parameters.
fstream	A type `basic_fstream` specialized on `char` template parameters.
ifstream	A type `basic_ifstream` specialized on `char` template parameters.
ofstream	A type `basic_ofstream` specialized on `char` template parameters.
wfstream	A type `basic_fstream` specialized on `wchar_t` template parameters.
wifstream	A type `basic_ifstream` specialized on `wchar_t` template parameters.
wofstream	A type `basic_ofstream` specialized on `wchar_t` template parameters.
wfilebuf	A type `basic_filebuf` specialized on `wchar_t` template parameters.

Classes

Class	Description
basic_filebuf	The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are determined by the class `Tr`, to and from a sequence of elements stored in an external file.

Class	Description
basic_fstream	The class template describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type `Elem`, whose character traits are determined by the class `Tr`.
basic_ifstream	The class template describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type `Elem`, whose character traits are determined by the class `Tr`.
basic_ofstream	The class template describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_filebuf<**Elem**, **Tr**>, with elements of type `Elem`, whose character traits are determined by the class `Tr`.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<fstream>` typedefs

Article • 12/06/2021

filebuf
fstream
ifstream
ofstream
wfilebuf
wfstream
wifstream
wofstream

`filebuf`

A type `basic_filebuf` specialized on `char` template parameters.

```
C++
```

```
typedef basic_filebuf<char, char_traits<char>> filebuf;
```

Remarks

The type is a synonym for class template basic_filebuf, specialized for elements of type `char` with default character traits.

`fstream`

A type `basic_fstream` specialized on `char` template parameters.

```
C++
```

```
typedef basic_fstream<char, char_traits<char>> fstream;
```

Remarks

The type is a synonym for class template basic_fstream, specialized for elements of type `char` with default character traits.

`ifstream`

Defines a stream to be used to read single-byte character data serially from a file. `ifstream` is a typedef that specializes the class template `basic_ifstream` for `char`.

There is also `wifstream`, a typedef that specializes `basic_ifstream` to read `wchar_t` double-wide characters. For more information, see wifstream.

C++

```
typedef basic_ifstream<char, char_traits<char>> ifstream;
```

Remarks

The type is a synonym for class template basic_ifstream, specialized for elements of type char with default character traits. An example is

C++

```
using namespace std;

ifstream infile("existingtextfile.txt");

if (!infile.bad())
{
    // Dump the contents of the file to cout.
    cout << infile.rdbuf();infile.close();
}
```

`ofstream`

A type `basic_ofstream` specialized on `char` template parameters.

C++

```
typedef basic_ofstream<char, char_traits<char>> ofstream;
```

Remarks

The type is a synonym for class template basic_ofstream, specialized for elements of type `char` with default character traits.

`wfstream`

A type `basic_fstream` specialized on `wchar_t` template parameters.

```
C++
```

```
typedef basic_fstream<wchar_t, char_traits<wchar_t>> wfstream;
```

Remarks

The type is a synonym for class template basic_fstream, specialized for elements of type `wchar_t` with default character traits.

`wifstream`

A type `basic_ifstream` specialized on `wchar_t` template parameters.

```
C++
```

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t>> wifstream;
```

Remarks

The type is a synonym for class template basic_ifstream, specialized for elements of type `wchar_t` with default character traits.

`wofstream`

A type `basic_ofstream` specialized on `wchar_t` template parameters.

```
C++
```

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t>> wofstream;
```

Remarks

The type is a synonym for class template basic_ofstream, specialized for elements of type `wchar_t` with default character traits.

`wfilebuf`

A type `basic_filebuf` specialized on `wchar_t` template parameters.

```
C++
```

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t>> wfilebuf;
```

Remarks

The type is a synonym for class template basic_filebuf, specialized for elements of type `wchar_t` with default character traits.

See also

<fstream>

basic_filebuf Class

Article • 06/09/2022

Describes a stream buffer that controls the transmission of elements of type *Char_T*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

Syntax

```
C++
```

```
template <class Char_T, class Tr = char_traits<Char_T>>
class basic_filebuf : public basic_streambuf<Char_T, Tr>
```

Parameters

Char_T
The basic element of the file buffer.

Tr
The traits of the basic element of the file buffer (usually `char_traits<Char_T>`).

Remarks

The class template describes a stream buffer that controls the transmission of elements of type *Char_T*, whose character traits are determined by the class *Tr*, to and from a sequence of elements stored in an external file.

> **Note**
>
> Objects of type `basic_filebuf` are created with an internal buffer of type **char*** regardless of the `char_type` specified by the type parameter *Char_T*. This means that a Unicode string (containing `wchar_t` characters) will be converted to an ANSI string (containing `char` characters) before it is written to the internal buffer. To store Unicode strings in the buffer, create a new buffer of type `wchar_t` and set it using the **basic_streambuf::pubsetbuf** `()` method. To see an example that demonstrates this behavior, see below.

An object of class `basic_filebuf<Char_T, Tr>` stores a file pointer, which designates the `FILE` object that controls the stream associated with an open file. It also stores pointers to two file conversion facets for use by the protected member functions overflow and underflow. For more information, see basic_filebuf::open.

Example

The following example demonstrates how to force an object of type `basic_filebuf<wchar_t>` to store Unicode characters in its internal buffer by calling the `pubsetbuf()` method.

C++

```
// unicode_basic_filebuf.cpp
// compile with: /EHsc

#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
#include <memory.h>
#include <string.h>

#define IBUFSIZE 16

using namespace std;

void hexdump(const string& filename);

int main()
{
    wchar_t* wszHello = L"Hello World";
    wchar_t wBuffer[128];

    basic_filebuf<wchar_t> wOutFile;

    // Open a file, wcHello.txt, then write to it, then dump the
    // file's contents in hex
    wOutFile.open("wcHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wcHello.txt\n";
        return -1;
    }
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "Hex Dump of wcHello.txt - note that output is ANSI chars:\n";
    hexdump(string("wcHello.txt"));
```

```
    // Open a file, wwHello.txt, then set the internal buffer of
    // the basic_filebuf object to be of type wchar_t, then write
    // to the file and dump the file's contents in hex
    wOutFile.open("wwHello.txt",
        ios_base::out | ios_base::trunc | ios_base::binary);
    if(!wOutFile.is_open())
    {
        cout << "Error Opening wwHello.txt\n";
        return -1;
    }
    wOutFile.pubsetbuf(wBuffer, (streamsize)128);
    wOutFile.sputn(wszHello, (streamsize)wcslen(wszHello));
    wOutFile.close();
    cout << "\nHex Dump of wwHello.txt - note that output is wchar_t
chars:\n";
    hexdump(string("wwHello.txt"));

    return 0;
}

// dump contents of filename to stdout in hex
void hexdump(const string& filename)
{
    fstream ifile(filename.c_str(),
        ios_base::in | ios_base::binary);
    char *ibuff = new char[IBUFSIZE];
    char *obuff = new char[(IBUFSIZE*2)+1];
    int i;

    if(!ifile.is_open())
    {
        cout << "Cannot Open " << filename.c_str()
             << " for reading\n";
        return;
    }
    if(!ibuff || !obuff)
    {
        cout << "Cannot Allocate buffers\n";
        ifile.close();
        return;
    }

    while(!ifile.eof())
    {
        memset(obuff,0,(IBUFSIZE*2)+1);
        memset(ibuff,0,IBUFSIZE);
        ifile.read(ibuff,IBUFSIZE);

        // corner case where file is exactly a multiple of
        // 16 bytes in length
        if(ibuff[0] == 0 && ifile.eof())
            break;

        for(i = 0; i < IBUFSIZE; i++)
        {
```

```
            if(ibuff[i] >= ' ')
                obuff[i] = ibuff[i];
            else
                obuff[i] = '.';

            cout << setfill('0') << setw(2) << hex
                 << (int)ibuff[i] << ' ';
        }
        cout << "  " << obuff << endl;
    }
    ifile.close();
}
```

Output

```
Hex Dump of wcHello.txt - note that output is ANSI chars:
48 65 6c 6c 6f 20 57 6f 72 6c 64 00 00 00 00 00   Hello World.....

Hex Dump of wwHello.txt - note that output is wchar_t chars:
48 00 65 00 6c 00 6c 00 6f 00 20 00 57 00 6f 00   H.e.l.l.o. .W.o.
72 00 6c 00 64 00 00 00 00 00 00 00 00 00 00 00   r.l.d...........
```

Constructors

Constructor	Description
basic_filebuf	Constructs an object of type `basic_filebuf`.

Typedefs

Type name	Description
char_type	Associates a type name with the `Char_T` template parameter.
int_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.
off_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.
pos_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the `Tr` scope.
traits_type	Associates a type name with the `Tr` template parameter.

Member functions

Member function	Description
close	Closes a file.
is_open	Indicates whether a file is open.
open	Opens a file.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	The protected virtual member function tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
setbuf	The protected virtual member function performs an operation particular to each derived stream buffer.
Swap	Exchanges the content of this `basic_filebuf` for the content of the provided `basic_filebuf` parameter.
sync	Protected, virtual function tries to synchronize the controlled streams with any associated external streams.
uflow	Protected, virtual function to extract the current element from the input stream.
underflow	Protected, virtual function to extract the current element from the input stream.

Requirements

Header: <fstream>

Namespace: std

basic_filebuf::basic_filebuf

Constructs an object of type `basic_filebuf`.

```
C++
```

```
basic_filebuf();

basic_filebuf(basic_filebuf&& right);
```

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores a null pointer in the file pointer.

The second constructor initializes the object with the contents of `right`, treated as an rvalue reference.

basic_filebuf::char_type

Associates a type name with the `Char_T` template parameter.

```
C++

typedef Char_T char_type;
```

basic_filebuf::close

Closes a file.

```
C++

basic_filebuf<Char_T, Tr> *close();
```

Return Value

The member function returns a null pointer if the file pointer is a null pointer.

Remarks

`close` calls `fclose(fp)`. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **`this`** to indicate that the file was successfully closed.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls overflow. It also inserts any sequence

needed to restore the initial conversion state, by using the file conversion facet `fac` to call `fac.unshift` as needed. Each produced element `byte` of type `char` is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(byte, fp)`. If the call to `fac.unshift` or any write fails, the function doesn't succeed.

Example

The following sample assumes two files in the current directory: *basic_filebuf_close.txt* (contents is "testing") and *iotest.txt* (contents is "ssss").

C++

```
// basic_filebuf_close.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main() {
   using namespace std;
   ifstream file;
   basic_ifstream <wchar_t> wfile;
   char c;
   // Open and close with a basic_filebuf
   file.rdbuf()->open( "basic_filebuf_close.txt", ios::in );
   file >> c;
   cout << c << endl;
   file.rdbuf( )->close( );

   // Open/close directly
   file.open( "iotest.txt" );
   file >> c;
   cout << c << endl;
   file.close( );

   // open a file with a wide character name
   wfile.open( L"iotest.txt" );

   // Open and close a nonexistent with a basic_filebuf
   file.rdbuf()->open( "ziotest.txt", ios::in );
   cout << file.fail() << endl;
   file.rdbuf( )->close( );

   // Open/close directly
   file.open( "ziotest.txt" );
   cout << file.fail() << endl;
   file.close( );
}
```

Output

```
t
s
0
1
```

basic_filebuf::int_type

Makes this type within `basic_filebuf` scope equivalent to the type of the same name in the `Tr` scope.

C++

```
typedef typename traits_type::int_type int_type;
```

basic_filebuf::is_open

Indicates whether a file is open.

C++

```
bool is_open() const;
```

Return Value

`true` if the file pointer isn't null.

Example

C++

```
// basic_filebuf_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;
   ifstream file;
   cout << boolalpha << file.rdbuf( )->is_open( ) << endl;
```

```
    file.open( "basic_filebuf_is_open.cpp" );
    cout << file.rdbuf( )->is_open( ) << endl;
}
```

Output

```
false
true
```

basic_filebuf::off_type

Makes this type within `basic_filebuf` scope equivalent to the type of the same name in the `Tr` scope.

C++

```
typedef typename traits_type::off_type off_type;
```

basic_filebuf::open

Opens a file.

C++

```
basic_filebuf<Char_T, Tr> *open(
    const char* filename,
    ios_base::openmode mode,
    int protection = (int)ios_base::_Openprot);

basic_filebuf<Char_T, Tr> *open(
    const char* filename,
    ios_base::openmode mode);

basic_filebuf<Char_T, Tr> *open(
    const wchar_t* filename,
    ios_base::openmode mode,
    int protection = (int)ios_base::_Openprot);

basic_filebuf<Char_T, Tr> *open(
    const wchar_t* filename,
    ios_base::openmode mode);
```

Parameters

filename
The name of the file to open.

mode
One of the enumerations in ios_base::openmode.

protection
The default file opening protection, equivalent to the *shflag* parameter in _fsopen, _wfsopen.

Return Value

If the buffer is already open, or if the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns `this`.

Remarks

This function uses a `FILE *` to back the `basic_filebuf` as though you had called fopen/wfopen`(filename, strmode)`. `strmode` is determined from `mode & ~(`ate `|` binary`)`:

- `ios_base::in` becomes `"r"` (open existing file for reading).
- ios_base::out or `ios_base::out | ios_base::trunc` becomes `"w"` (truncate existing file or create for writing).
- `ios_base::out | app` becomes `"a"` (open existing file for appending all writes).
- `ios_base::in | ios_base::out` becomes `"r+"` (open existing file for reading and writing).
- `ios_base::in | ios_base::out | ios_base::trunc` becomes `"w+"` (truncate existing file or create for reading and writing).
- `ios_base::in | ios_base::out | ios_base::app` becomes `"a+"` (open existing file for reading and for appending all writes).

If `mode & ios_base::binary` is nonzero, the function appends `b` to `strmode` to open a binary stream instead of a text stream. If `mode & ios_base::ate` is nonzero and the file was successfully opened, the current location in the stream is positioned at the end of file. If that fails, the file is closed.

If the above operations completed successfully, the file conversion facet is determined: `use_facet<codecvt<Char_T, char, traits_type::`state_type`> >(`getloc`)`, for use by underflow and overflow.

If the file couldn't be successfully opened, `nullptr` is returned.

Example

See basic_filebuf::close for an example that uses `open`.

basic_filebuf::operator=

Assign the content of this stream buffer object. This is a move assignment involving an rvalue that doesn't leave a copy behind.

```
C++
```

```
basic_filebuf& operator=(basic_filebuf&& right);
```

Parameters

right
An rvalue reference to a basic_filebuf object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of `right`, treated as an rvalue reference. For more information, see Rvalue reference declarator: &&.

basic_filebuf::overflow

Called when a new character is inserted into a full buffer.

```
C++
```

```
virtual int_type overflow(int_type _Meta = traits_type::eof);
```

Parameters

_Meta
The character to insert into the buffer or `traits_type::eof`.

Return Value

If the function can't succeed, it returns `traits_type::eof`. Otherwise, it returns `traits_type::` not_eof `(_Meta)`.

Remarks

If `_Meta != traits_type::` eof, the protected virtual member function attempts to insert the element `ch = traits_type::` to_char_type `(_Meta)` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by `ch`, by using the file conversion facet `fac` to call `fac.out` as needed. Each produced element `ch` of type *char* is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(ch, fp)`. If any conversion or write fails, the function doesn't succeed.

basic_filebuf::pbackfail

Tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
C++
```

```
virtual int_type pbackfail(int_type _Meta = traits_type::eof);
```

Parameters

_Meta
The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function can't succeed, it returns `traits_type::eof`. Otherwise, it returns `traits_type::not_eof(_Meta)`.

Remarks

The protected virtual member function puts back an element into the input buffer and then makes it the current element (pointed to by the next pointer). If `_Meta == traits_type::eof`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `ch = traits_type::to_char_type(_Meta)`. The function can put back an element in various ways:

- If a `putback` position is available, and the element stored there compares equal to `ch`, it can decrement the next pointer for the input buffer.
- If the function can make a `putback` position available, it can do so, set the next pointer to point at that position, and store `ch` in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling `ungetc` for an element of type `char`.

basic_filebuf::pos_type

Makes this type within `basic_filebuf` scope equivalent to the type of the same name in the `Tr` scope.

```
C++
```

```
typedef typename traits_type::pos_type pos_type;
```

basic_filebuf::seekoff

Tries to alter the current positions for the controlled streams.

```
C++
```

```
virtual pos_type seekoff(
    off_type _Off,
```

```
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off
The position to seek for relative to *_Way*.

_Way
The starting point for offset operations. See seekdir for possible values.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position.

Remarks

The protected virtual member function attempts to alter the current positions for the controlled streams. For an object of class basic_filebuf`<Char_T, Tr>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero refers to the first element of the stream. (An object of type pos_type stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams, binary streams, and wide streams.

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it attempts to alter the stream position by calling `fseek(fp, _Off, _Way)`. If that function succeeds and the resulting position `fposn` can be determined by calling `fgetpos(fp, &fposn)`, the function succeeds. If the function succeeds, it returns a value of type `pos_type` containing `fposn`. Otherwise, it returns an invalid stream position.

basic_filebuf::seekpos

Tries to alter the current positions for the controlled streams.

C++

```
virtual pos_type seekpos(
    pos_type _Sp,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Sp
The position to seek for.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it attempts to alter the stream position by calling `fsetpos(fp, &fposn)`, where `fposn` is the `fpos_t` object stored in `pos`. If that function succeeds, the function returns `pos`. Otherwise, it returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function attempts to alter the current positions for the controlled streams. For an object of class basic_filebuf`<Char_T, Tr>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero refers to the first element of the stream. (An object of type `pos_type` stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either pubseekoff or pubseekpos. Calls to `pubseekoff` (and to `seekoff`) have various limitations for text streams, binary streams, and wide streams.

For a wide stream, if any insertions have occurred since the stream was opened, or since the last call to `streampos`, the function calls overflow. It also inserts any sequence needed to restore the initial conversion state, by using the file conversion facet `fac` to call `fac.unshift` as needed. Each produced element `byte` of type **`char`** is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form

`fputc(byte, fp)`. If the call to `fac.unshift` or any write fails, the function doesn't succeed.

basic_filebuf::setbuf

Performs an operation particular to each derived stream buffer.

```
C++
```

```
virtual basic_streambuf<Char_T, Tr> *setbuf(
    char_type* _Buffer,
    streamsize count);
```

Parameters

_Buffer
Pointer to a buffer.

count
Size of the buffer.

Return Value

The protected member function returns zero if the file pointer `fp` is a null pointer.

Remarks

`setbuf` calls `setvbuf( fp, (char*) _Buffer, _IOFBF, count * sizeof( Char_T))` to offer the array of `count` elements beginning at *_Buffer* as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns **`this`** to signal success.

basic_filebuf::swap

Exchanges the contents of this `basic_filebuf` for the contents of the provided `basic_filebuf`.

```
C++
```

```
void swap(basic_filebuf& right);
```

Parameters

right
An lvalue reference to another `basic_filebuf`.

basic_filebuf::sync

Tries to synchronize the controlled streams with any associated external streams.

```
C++
```

```
virtual int sync();
```

Return Value

Returns zero if the file pointer `fp` is a null pointer. Otherwise, it returns zero only if calls to both overflow and `fflush(fp)` succeed in flushing any pending output to the stream.

basic_filebuf::traits_type

Associates a type name with the `Tr` template parameter.

```
C++
```

```
typedef Tr traits_type;
```

basic_filebuf::underflow

Extracts the current element from the input stream.

```
C++
```

```
virtual int_type underflow();
```

Return Value

If the function can't succeed, it returns `traits_type::`eof. Otherwise, it returns `ch`, converted as described in the Remarks section.

Remarks

The protected virtual member function attempts to extract the current element `ch` from the input stream, and return the element as `traits_type::`to_int_type`(ch)`. It can do so in various ways:

- If a read position is available, it takes `ch` as the element stored in the read position and advances the next pointer for the input buffer.
- It can read one or more elements of type **`char`**, as if by successive calls of the form `fgetc(fp)`, and convert them to an element `ch` of type `Char_T` by using the file conversion facet `fac` to call `fac.in` as needed. If any read or conversion fails, the function doesn't succeed.

See also

<fstream>
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

basic_fstream Class

Article • 08/03/2021

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_filebuf< `Elem`, `Tr` >, with elements of type `Elem`, whose character traits are determined by the class `Tr`.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_fstream : public basic_iostream<Elem, Tr>
```

Parameters

Elem
The basic element of the file buffer.

Tr
The traits of the basic element of the file buffer (usually `char_traits`< `Elem` >).

Remarks

The object stores an object of class `basic_filebuf`< `Elem`, `Tr` >.

> **Note**
>
> The get pointer and put pointer of an fstream object are **NOT** independent of each other. If the get pointer moves, so does the put pointer.

Example

The following example demonstrates how to create a `basic_fstream` object that can be read from and written to.

C++

```cpp
// basic_fstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    fstream fs("fstream.txt", ios::in | ios::out | ios::trunc);
    if (!fs.bad())
    {
        // Write to the file.
        fs << "Writing to a basic_fstream object..." << endl;
        fs.close();

        // Dump the contents of the file to cout.
        fs.open("fstream.txt", ios::in);
        cout << fs.rdbuf();
        fs.close();
    }
}
```

Output

```
Writing to a basic_fstream object...
```

Constructors

Constructor	Description
basic_fstream	Constructs an object of type `basic_fstream`.

Member functions

Member function	Description
close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer, of type pointer to basic_filebuf< `Elem`, `Tr` >.

Member function	Description
swap	Exchanges the content of this object with the content of another `basic_fstream` object.

Requirements

Header: <fstream>

Namespace: std

basic_fstream::basic_fstream

Constructs an object of type `basic_fstream`.

C++

```
basic_fstream();

explicit basic_fstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_fstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

basic_fstream(basic_fstream&& right);
```

Parameters

_Filename
The name of the file to open.

_Mode
One of the enumerations in ios_base::openmode.

_Prot
The default file opening protection, equivalent to the *shflag* parameter in _fsopen, _wfsopen.

Remarks

The first constructor initializes the base class by calling basic_iostream(`sb`), where `sb` is the stored object of class basic_filebuf< **Elem**, **Tr**>. It also initializes `sb` by calling `basic_filebuf`< **Elem**, **Tr**>.

The second and third constructors initializes the base class by calling `basic_iostream`(**sb**). It also initializes `sb` by calling `basic_filebuf`< **Elem**, **Tr**>, and then **sb**.open(_ *Filename*, `_Mode`). If the latter function returns a null pointer, the constructor calls setstate(`failbit`).

The fourth constructor initializes the object with the contents of `right`, treated as an rvalue reference.

Example

See streampos for an example that uses `basic_fstream`.

basic_fstream::close

Closes a file.

```
C++
```

```
void close();
```

Remarks

The member function calls rdbuf-> close.

Example

See basic_filebuf::close for an example of how to use `close`.

basic_fstream::is_open

Determines if a file is open.

```
C++
```

```
bool is_open() const;
```

Return Value

`true` if the file is open, `false` otherwise.

Remarks

The member function returns rdbuf->is_open.

Example

See basic_filebuf::is_open for an example of how to use `is_open`.

basic_fstream::open

Opens a file.

C++

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in | ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

_Filename
The name of the file to open.

_Mode
One of the enumerations in ios_base::openmode.

_Prot
The default file opening protection, equivalent to the *shflag* parameter in _fsopen, _wfsopen.

Remarks

The member function calls rdbuf -> open(_ *Filename*, `_Mode`). If that function returns a null pointer, the function calls setstate(`failbit`).

Example

See basic_filebuf::open for an example of how to use `open`.

basic_fstream::operator=

Assigns to this object the content from a specified stream object. This is a move assignment that involves an rvalue that does not leave a copy behind.

```
C++

basic_fstream& operator=(basic_fstream&& right);
```

Parameters

right
An lvalue reference to a `basic_fstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *right*, treated as an rvalue reference.

basic_fstream::rdbuf

Returns the address of the stored stream buffer, of type pointer to basic_filebuf< **Elem**, **Tr**>.

```
C++
```

```
basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

The address of the stored stream buffer.

Example

See basic_filebuf::close for an example of how to use `rdbuf`.

basic_fstream::swap

Exchanges the contents of two `basic_fstream` objects.

```
C++
```

```
void swap(basic_fstream& right);
```

Parameters

right
An `lvalue` reference to a `basic_fstream` object.

Remarks

The member function exchanges the contents of this object and the contents of *right*.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`basic_ifstream` Class

Article • 06/09/2022

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_filebuf<Elem, Tr>`, with elements of type `Elem`, whose character traits are determined by the class `Tr`.

For more information, see basic_filebuf.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ifstream : public basic_istream<Elem, Tr>
```

Parameters

`Elem`
The basic element of the file buffer.

`Tr`
The traits of the basic element of the file buffer (usually `char_traits<Elem>`).

Remarks

The object stores an object of class `basic_filebuf<Elem, Tr>`.

Example

The following example shows how to read in text from a file.

C++

```
// basic_ifstream_class.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;
```

```
int main(int argc, char **argv)
{
    ifstream ifs("basic_ifstream_class.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
        ifs.close();
    }
}
```

Input: `basic_ifstream_class.txt`

```
C++
```

```
This is the contents of basic_ifstream_class.txt.
```

Output

```
C++
```

```
This is the contents of basic_ifstream_class.txt.
```

Constructors

Constructor	Description
basic_ifstream	Initializes a new instance of a `basic_ifstream` object.

Member functions

Member function	Description
close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer.

Member function	Description
swap	Exchanges the content of this `basic_ifstream` for the content of the provided `basic_ifstream`.

Operators

Operator	Description
operator=	Assigns the content of this stream object. This is a move assignment involving an `rvalue` that doesn't leave a copy behind.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_ifstream::basic_ifstream`

Constructs an object of type `basic_ifstream`.

C++

```
basic_ifstream();

explicit basic_ifstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_ifstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

basic_ifstream(basic_ifstream&& right);
```

Parameters

`_Filename`

The name of the file to open.

_Mode

One of the enumerations in ios_base::openmode.

_Prot

The default file opening protection, equivalent to the `shflag` parameter in _fsopen, _wfsopen.

Remarks

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_filebuf<Elem, Tr>`. It also initializes `sb` by calling `basic_filebuf<Elem, Tr>`.

The second and third constructors initialize the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_filebuf<Elem, Tr>`, then `sb.open(_Filename, _Mode | ios_base::in)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

The fourth constructor initializes the object with the contents of `right`, treated as an `rvalue` reference.

For more information, see basic_istream, basic_filebuf, setstate, and open.

Example

The following example shows how to read in text from a file. To create the file, see the example for basic_ofstream::basic_ofstream.

C++

```
// basic_ifstream_ctor.cpp
// compile with: /EHsc

#include <fstream>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    ifstream ifs("basic_ifstream_ctor.txt");
    if (!ifs.bad())
    {
        // Dump the contents of the file to cout.
        cout << ifs.rdbuf();
```

```
        ifs.close();
    }
}
```

`basic_ifstream::close`

Closes a file.

C++

```
void close();
```

Remarks

The member function calls rdbuf -> close.

Example

See basic_filebuf::close for an example that uses `close`.

`basic_ifstream::is_open`

Determines if a file is open.

C++

```
bool is_open() const;
```

Return Value

`true` if the file is open, **`false`** otherwise.

Remarks

The member function returns rdbuf -> is_open.

Example

See basic_filebuf::is_open for an example that uses `is_open`.

`basic_ifstream::open`

Opens a file.

C++

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::in,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

`_Filename`

The name of the file to open.

`_Mode`

One of the enumerations in ios_base::openmode.

`_Prot`

The default file opening protection, equivalent to the `shflag` parameter in _fsopen, _wfsopen.

Remarks

The member function calls `rdbuf->open(_Filename, _Mode | ios_base::in)`. For more information, see rdbuf and basic_filebuf::open. If open fails, the function calls `setstate(failbit)`, which may throw an `ios_base::failure` exception. For more information, see setstate.

Example

See basic_filebuf::open for an example that uses `open`.

`basic_ifstream::operator=`

Assigns the content of this stream object. This is a move assignment involving an `rvalue` that doesn't leave a copy behind.

```
C++

basic_ifstream& operator=(basic_ifstream&& right);
```

Parameters

`right`
An `rvalue` reference to a `basic_ifstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *`right`*, treated as an `rvalue` reference. For more information, see Lvalues and Rvalues.

`basic_ifstream::rdbuf`

Returns the address of the stored stream buffer.

```
C++

basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

A pointer to a basic_filebuf object representing the stored stream buffer.

Example

See basic_filebuf::close for an example that uses `rdbuf`.

`basic_ifstream::swap`

Exchanges the contents of two `basic_ifstream` objects.

```
C++
```

```
void swap(basic_ifstream& right);
```

Parameters

`right`

A reference to another stream buffer.

Remarks

The member function exchanges the contents of this object for the contents of *`right`*.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`basic_ofstream` Class

Article • 06/16/2022

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_filebuf< Elem, Tr>`, with elements of type `Elem`, whose character traits are determined by the class `Tr`. For more information, see basic_filebuf.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ofstream : public basic_ostream<Elem, Tr>
```

Parameters

`Elem`

The basic element of the file buffer.

`Tr`

The traits of the basic element of the file buffer (usually `char_traits<Elem>`).

Remarks

When the **`wchar_t`** specialization of `basic_ofstream` writes to the file, if the file is opened in text mode it will write an MBCS sequence. The internal representation will use a buffer of **`wchar_t`** characters.

The object stores an object of class `basic_filebuf< Elem, Tr>`.

Example

The following example shows how to create a `basic_ofstream` object and write text to it.

C++

```
// basic_ofstream_class.cpp
// compile with: /EHsc
#include <fstream>
```

```
using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

Constructors

Constructor	Description
basic_ofstream	Creates an object of type `basic_ofstream`.

Member functions

Member function	Description
close	Closes a file.
is_open	Determines if a file is open.
open	Opens a file.
rdbuf	Returns the address of the stored stream buffer.
swap	Exchange the contents of this `basic_ofstream` for the contents of the provided `basic_ofstream`.

Operators

Operator	Description
operator=	Assigns the content of this stream object. This is a move assignment involving an `rvalue reference` that doesn't leave a copy behind.

Requirements

Header: `<fstream>`

Namespace: `std`

`basic_ofstream::basic_ofstream`

Creates an object of type `basic_ofstream`.

C++

```
basic_ofstream();

explicit basic_ofstream(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

explicit basic_ofstream(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

basic_ofstream(
    basic_ofstream&& right);
```

Parameters

`_Filename`
The name of the file to open.

`_Mode`
One of the enumerations in ios_base::openmode.

`_Prot`
The default file opening protection, equivalent to the `shflag` parameter in _fsopen, _wfsopen.

`right`
The rvalue reference to the `basic_ofstream` object being used to initialize this `basic_ofstream` object.

Remarks

The first constructor initializes the base class by calling `basic_ostream(sb)`, where `sb` is the stored object of class `basic_filebuf< Elem, Tr>`. It also initializes `sb` by calling `basic_filebuf < Elem, Tr >`.

The second and third constructors initialize the base class by calling `basic_ostream( sb)`. It also initializes `sb` by calling `basic_filebuf`< `Elem`, `Tr`> and then `sb.open( _Filename, _Mode | ios_base::out)`. If the latter function returns a `NULL` pointer, the constructor calls `setstate(failbit)`.

The fourth constructor is a copy function. It initializes the object with the contents of *`right`*, treated as an rvalue reference.

For more information, see basic_ostream, open, and setstate.

Example

The following example shows how to create a `basic_ofstream` object and write text to it.

C++

```
// basic_ofstream_ctor.cpp
// compile with: /EHsc
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream ofs("C:\\ofstream.txt");
    if (!ofs.bad())
    {
        ofs << "Writing to a basic_ofstream object..." << endl;
        ofs.close();
    }
}
```

`basic_ofstream::close`

Closes a file.

C++

```
void close();
```

Remarks

The member function calls `rdbuf->close`. For more information, see rdbuf and close.

Example

See basic_filebuf::close for an example that uses `close`.

`basic_ofstream::is_open`

Indicates whether a file is open.

C++

```
bool is_open() const;
```

Return Value

`true` if the file is open, **`false`** otherwise.

Remarks

The member function returns `rdbuf->is_open`. For more information, see rdbuf and is_open.

Example

C++

```
// basic_ofstream_is_open.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;
   ifstream file;
   // Open and close with a basic_filebuf
   file.rdbuf( )->open( "basic_ofstream_is_open.txt", ios::in );
   file.close( );
   if (file.is_open())
      cout << "it's open" << endl;
   else
      cout << "it's closed" << endl;
}
```

`basic_ofstream::open`

Opens a file.

```
C++
```

```
void open(
    const char* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const char* _Filename,
    ios_base::openmode _Mode);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode = ios_base::out,
    int _Prot = (int)ios_base::_Openprot);

void open(
    const wchar_t* _Filename,
    ios_base::openmode _Mode);
```

Parameters

`_Filename`

The name of the file to open.

`_Mode`

One of the enumerations in ios_base::openmode.

`_Prot`

The default file opening protection, equivalent to the `shflag` parameter in _fsopen, _wfsopen.

Remarks

The member function calls `rdbuf -> open(_ Filename, _Mode | ios_base::out)`. If that function returns a `NULL` pointer, the function calls `setstate(failbit)`.

For more information, see rdbuf, open, and setstate.

Example

See basic_filebuf::open for an example that uses `open`.

`basic_ofstream::operator=`

Assigns the content of this stream object. This is a move assignment involving an `rvalue reference` that doesn't leave a copy behind.

```
C++

basic_ofstream& operator=(basic_ofstream&& right);
```

Parameters

`right`
An rvalue reference to a `basic_ofstream` object.

Return Value

Returns `*this`.

Remarks

The member operator replaces the contents of the object by using the contents of *`right`*, treated as an rvalue reference.

`basic_ofstream::rdbuf`

Returns the address of the stored stream buffer.

```
C++

basic_filebuf<Elem, Tr> *rdbuf() const
```

Return Value

Returns the address of the stored stream buffer.

Example

See basic_filebuf::close for an example that uses `rdbuf`.

`basic_ofstream::swap`

Exchanges the contents of two `basic_ofstream` objects.

```
C++
```

```
void swap(basic_ofstream& right);
```

Parameters

`right`
An `lvalue` reference to another `basic_ofstream` object.

Remarks

The member function exchanges the contents of this object for the contents of *`right`*.

See also

basic_ostream Class
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<functional>`

Article • 05/20/2022

Defines C++ Standard Library functions that help construct *function objects*, also known as *functors*, and their binders. A function object is an object of a type that defines `operator()`. A function object can be a function pointer, but more typically, the object is used to store additional information that can be accessed during a function call.

Requirements

Header: <functional>

Namespace: std

Remarks

Algorithms require two types of function objects: *unary* and *binary*. Unary function objects require one argument, and binary function objects require two arguments. A function object and function pointers can be passed as a predicate to an algorithm, but function objects are also adaptable and increase the scope, flexibility, and efficiency of the C++ Standard Library. If, for example, a value needed to be bound to a function before being passed to an algorithm, then a function pointer could not be used. Function adaptors convert function pointers into adaptable function objects that can be bound to a value. The header <functional> also contains member function adaptors that allow member functions to be called as adaptable function objects. Functions are adaptable if they have nested type declarations specifying their argument and return types. Function objects and their adaptors allow the C++ Standard Library to upgrade existing applications and help integrate the library into the C++ programming environment.

The implementation of the function objects in <functional> includes *transparent operator functors*, which are specializations of standard function objects and take no template parameters, and perform perfect forwarding of the function arguments and perfect return of the result. These template specializations do not require that you specify argument types when you invoke arithmetic, comparison, logical, and bitwise operator functors. You can overload arithmetic, comparison, logical, or bitwise operators for your own types, or for heterogeneous combinations of types, and then use the transparent operator functors as function arguments. For example, if your type *MyType* implements `operator<`, you can call `sort(my_collection.begin(), my_collection.end(),`

`less<>())` instead of explicitly specifying the type `sort(my_collection.begin(), my_collection.end(), less<MyType>())`.

The following features are added in C++11, C++14 and C++17:

- A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
- A *callable type* is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.
- A *callable object* is an object of a callable type.
- A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.
- A *call wrapper* is an object of a call wrapper type.
- A *target object* is the callable object held by a call wrapper object.

The pseudo-function `INVOKE(f, t1, t2, ..., tN)` means one of the following things:

- `(t1.*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`.
- `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is not one of the types described in the previous item.
- `t1.*f` when N == 1 and `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`.
- `(*t1).*f` when N == 1 and `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item.
- `f(t1, t2, ..., tN)` in all other cases.

The pseudo-function `INVOKE(f, t1, t2, ..., tN, R)` means `INVOKE(f, t1, t2, ..., tN)` implicitly converted to `R`.

If a call wrapper has a *weak result type*, the type of its member type `result_type` is based on the type `T` of the target object of the wrapper, as follows:

- If `T` is a pointer to function, `result_type` is a synonym for the return type of `T`.
- If `T` is a pointer to member function, `result_type` is a synonym for the return type of `T`.
- If `T` is a class type that has a member type `result_type`, then `result_type` is a synonym for `T::result_type`.
- Otherwise, there is no member `result_type`.

Every call wrapper has a move constructor and a copy constructor. A *simple call wrapper* is a call wrapper that has an assignment operator and whose copy constructor, move constructor, and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called by using an arbitrary argument list and that delivers the arguments to the wrapped callable object as references. All rvalue arguments are delivered as rvalue references, and lvalue arguments are delivered as lvalue references.

Members

Classes

Name	Description
bad_function_call	A class that describes an exception thrown to indicate that a call to `operator()` on a function object failed because the object was empty.
binary_negate	A class template providing a member function that negates the return value of a specified binary function. (Deprecated in C++17.)
binder1st	A class template providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
binder2nd	A class template providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
boyer_moore_horspool_searcher	
boyer_moore_searcher	

Name	Description
const_mem_fun_ref_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun_t	An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun1_ref_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
const_mem_fun1_t	An adapter class that allows a const member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
default_searcher	
function	A class that wraps a callable object.
hash	A class that computes a hash code for a value.
is_bind_expression	A class that tests if a particular type is generated by calling `bind`.
is_placeholder	A class that tests if a particular type is a placeholder.
mem_fun_ref_t	An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)
mem_fun_t	An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
mem_fun1_ref_t	An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a reference argument. (Deprecated in C++11, removed in C++17.)

Name	Description
mem_fun1_t	An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. (Deprecated in C++11, removed in C++17.)
pointer_to_binary_function	Converts a binary function pointer into an adaptable binary function. (Deprecated in C++11, removed in C++17.)
pointer_to_unary_function	Converts a unary function pointer into an adaptable unary function. (Deprecated in C++11, removed in C++17.)
reference_wrapper	A class that wraps a reference.
unary_negate	A class template providing a member function that negates the return value of a specified unary function. (Deprecated in C++17.)

Functions

Name	Description
bind	Binds arguments to a callable object.
bind1st	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the first argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
bind2nd	A helper template function that creates an adaptor to convert a binary function object into a unary function object by binding the second argument of the binary function to a specified value. (Deprecated in C++11, removed in C++17.)
bit_and	Returns the bitwise AND (binary `operator&`) of the two parameters.
bit_not	Returns the bitwise complement (`operator~`) of the parameter. (Added in C++14.)
bit_or	Returns the bitwise OR (`operator|`) of the two parameters.
bit_xor	Returns the bitwise XOR (`operator^`) of the two parameters.
cref	Constructs a const `reference_wrapper` from an argument.
invoke	

Name	Description
mem_fn	Generates a simple call wrapper.
mem_fun	Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments. (Deprecated in C++11, removed in C++17.)
mem_fun_ref	A helper template function used to construct function object adaptors for member functions when initialized with reference arguments.
not1	Returns the complement of a unary predicate. (Deprecated in C++17.)
not2	Returns the complement of a binary predicate. (Deprecated in C++17.)
not_fn	Returns the complement of the result of its function object. (Added in C++17.)
ptr_fun	A helper template function used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions. (Deprecated in C++11, removed in C++17.)
ref	Constructs a `reference_wrapper` from an argument.
swap	Swaps two `function` objects.

Structs

Name	Description
binary_function	An empty base class that defines types that may be inherited by derived class that provides a binary function object. (Deprecated in C++11, removed in C++17.)
divides	The class provides a predefined function object that performs the arithmetic operation of division on elements of a specified value type.
equal_to	A binary predicate that tests whether a value of a specified type is equal to another value of that type.
greater	A binary predicate that tests whether a value of a specified type is greater than another value of that type.
greater_equal	A binary predicate that tests whether a value of a specified type is greater than or equal to another value of that type.
less	A binary predicate that tests whether a value of a specified type is less than another value of that type.

Name	Description
less_equal	A binary predicate that tests whether a value of a specified type is less than or equal to another value of that type.
logical_and	The class provides a predefined function object that performs the logical operation of conjunction on elements of a specified value type and tests for the truth or falsity of the result.
logical_not	The class provides a predefined function object that performs the logical operation of negation on elements of a specified value type and tests for the truth or falsity of the result.
logical_or	The class provides a predefined function object that performs the logical operation of disjunction on elements of a specified value type and tests for the truth or falsity of the result.
minus	The class provides a predefined function object that performs the arithmetic operation of subtraction on elements of a specified value type.
modulus	The class provides a predefined function object that performs the arithmetic operation of modulus on elements of a specified value type.
multiplies	The class provides a predefined function object that performs the arithmetic operation of multiplication on elements of a specified value type.
negate	The class provides a predefined function object that returns the negative of an element value.
not_equal_to	A binary predicate that tests whether a value of a specified type is not equal to another value of that type.
plus	The class provides a predefined function object that performs the arithmetic operation of addition on elements of a specified value type.
unary_function	An empty base class that defines types that may be inherited by derived class that provides a unary function object. (Deprecated in C++11, removed in C++17.)

Objects

Name	Description
_1.._M	Placeholders for replaceable arguments.

Operators

Name	Description

Name	Description
operator==	Disallows equality comparison of callable objects.
operator!=	Disallows inequality comparison of callable objects.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<functional>` functions

Article • 12/13/2021

These functions are deprecated in C++11 and removed in C++17:

bind1st
bind2nd
mem_fun
mem_fun_ref
ptr_fun

These functions are deprecated in C++17:

not1
not2

`bind`

Binds arguments to a callable object.

C++

```
template <class FT, class T1, class T2, ..., class TN>
    unspecified bind(FT fn, T1 t1, T2 t2, ..., TN tN);

template <class RTy, class FT, class T1, class T2, ..., class TN>
    unspecified bind(FT fn, T1 t1, T2 t2, ..., TN tN);
```

Parameters

FT

The type of the object to call. For example, the type of the function, function object, function pointer/reference, or member function pointer.

RTy

The return type. When specified, it will be the return type of the bound call. Otherwise, the return type is the return type of `FT`.

TN

The type of the Nth call argument.

fn

The object to call.

tN

The Nth call argument.

Remarks

The types `FT, T1, T2, ..., TN` must be copy-constructible, and `INVOKE(fn, t1, ..., tN)` must be a valid expression for some values `w1, w2, ..., wN`.

The first template function returns a forwarding call wrapper `g` with a weak result type. The effect of `g(u1, u2, ..., uM)` is `INVOKE(f, v1, v2, ..., vN,` invoke_result`<FT cv (V1, V2, ..., VN)>::type)`, where `cv` is the cv-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. You use it to bind arguments to a callable object to make a callable object with a tailored argument list.

The second template function returns a forwarding call wrapper `g` with a nested type `result_type` that is a synonym for `RTy`. The effect of `g(u1, u2, ..., uM)` is `INVOKE(f, v1, v2, ..., vN, RTy)`, where `cv` is the cv-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below. You use it to bind arguments to a callable object to make a callable object with a tailored argument list and with a specified return type.

The values of the bound arguments `v1, v2, ..., vN` and their corresponding types `V1, V2, ..., VN` depend on the type of the corresponding argument `ti` of type `Ti` in the call to `bind` and the cv-qualifiers `cv` of the call wrapper `g` as follows:

If `ti` is of type `reference_wrapper<T>` the argument `vi` is `ti.get()` and its type `Vi` is `T&`;

If the value of `std::is_bind_expression<Ti>::value` is **`true`** the argument `vi` is `ti(u1, u2, ..., uM)` and its type `Vi` is `result_of<Ti cv (U1&, U2&, ..., UN&>::type`;

If the value `j` of `std::is_placeholder<Ti>::value` isn't zero the argument `vi` is `uj` and its type `Vi` is `Uj&`;

Otherwise the argument `vi` is `ti` and its type `Vi` is `Ti cv &`.

For example, given a function `f(int, int)` the expression `bind(f, _1, 0)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(x, 0)`. The expression `bind(f, 0,`

`_1)` returns a forwarding call wrapper `cw` such that `cw(x)` calls `f(0, x)`.

The number of arguments in a call to `bind` and the argument `fn` must be equal to the number of arguments that can be passed to the callable object `fn`. For example, `bind(cos, 1.0)` is correct, and both `bind(cos)` and `bind(cos, _1, 0.0)` are incorrect.

The number of arguments in the function call to the call wrapper returned by `bind` must be at least as large as the highest numbered value of `is_placeholder<PH>::value` for all of the placeholder arguments in the call to `bind`. For example, `bind(cos, _2)(0.0, 1.0)` is correct (and returns `cos(1.0)`), and `bind(cos, _2)(0.0)` is incorrect.

Example

C++

```
// std__functional__bind.cpp
// compile with: /EHsc
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std::placeholders;

void square(double x)
{
    std::cout << x << "^2 == " << x * x << std::endl;
}

void product(double x, double y)
{
    std::cout << x << "*" << y << " == " << x * y << std::endl;
}

int main()
{
    double arg[] = { 1, 2, 3 };

    std::for_each(&arg[0], arg + 3, square);
    std::cout << std::endl;

    std::for_each(&arg[0], arg + 3, std::bind(product, _1, 2));
    std::cout << std::endl;

    std::for_each(&arg[0], arg + 3, std::bind(square, _1));

    return (0);
}
```

Output

```
1^2 == 1
2^2 == 4
3^2 == 9

1*2 == 2
2*2 == 4
3*2 == 6

1^2 == 1
2^2 == 4
3^2 == 9
```

`bind1st`

A helper template function that creates an adaptor to convert a binary function object into a unary function object. It binds the first argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

C++

```
template <class Operation, class Type>
    binder1st <Operation> bind1st (const Operation& func, const Type& left);
```

Parameters

`func`

The binary function object to be converted to a unary function object.

`left`

The value to which the first argument of the binary function object is to be bound.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *`left`*.

Remarks

Function binders are a kind of function adaptor. Because they return function objects, they can be used in certain types of function composition to construct more complicated and powerful expressions.

If *func* is an object of type `Operation` and `c` is a constant, then `bind1st( func, c )` is the same as the binder1st class constructor `binder1st<Operation>(func, c)`, and is more convenient to use.

Example

C++

```
// functional_bind1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan5: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 5);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind1st(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1a << "." << endl;

    // Compare: counting the number of integers > 5 in the vector
    // with a user defined function object
```

```
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan5());
    cout << "The number of elements in v1 greater than 5 is: "
         << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind2nd(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 5 is: 4.
The number of elements in v1 less than 10 is: 2.
```

`bind2nd`

A helper template function that creates an adaptor to convert a binary function object into a unary function object. It binds the second argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

C++

```
template <class Operation, class Type>
    binder2nd <Operation> bind2nd(const Operation& func, const Type& right);
```

Parameters

`func`

The binary function object to be converted to a unary function object.

`right`

The value to which the second argument of the binary function object is to be bound.

Return Value

The unary function object result of binding the second argument of the binary function object to *`right`*.

Remarks

Function binders are a kind of function adaptor. Because they return function objects, they can be used in certain types of function composition to construct more complicated and powerful expressions.

If `func` is an object of type `Operation` and `c` is a constant, then `bind2nd(func, c)` is the same as the binder2nd class constructor `binder2nd<Operation>(func, c)`, and more convenient to use.

Example

C++

```
// functional_bind2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan15: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
        return (result_type)(i > 15);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;
```

```
    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1a;
    result1a = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1a << "." << endl;

    // Compare counting the number of integers > 15 in the vector
    // with a user-defined function object
    vector<int>::iterator::difference_type result1b;
    result1b = count_if(v1.begin(), v1.end(), greaterthan15());
    cout << "The number of elements in v1 greater than 15 is: "
         << result1b << "." << endl;

    // Count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(), bind1st(greater<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 greater than 15 is: 2.
The number of elements in v1 less than 10 is: 2.
```

bit_and

A predefined function object that does a bitwise AND operation (binary `operator&`) on its arguments.

C++

```
template <class Type = void>
struct bit_and : public binary_function<Type, Type, Type
{
    Type operator()(
    const Type& Left,
    const Type& Right) const;
};

// specialized transparent functor for operator&
template <>
struct bit_and<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const  ->
        decltype(std::forward<T>(Left) & std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*

Any type that supports an operator& that takes operands of the specified or inferred types.

Left

The left operand of the bitwise AND operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the bitwise AND operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of Left & Right. The specialized template does perfect forwarding of the result, which has the type that's returned by operator&.

Remarks

The bit_and functor is restricted to integral types for the basic data types, or to user-defined types that implement binary operator&.

bit_not

A predefined function object that does a bitwise complement (NOT) operation (unary operator~) on its argument. Added in C++14.

C++

```
template <class Type = void>
struct bit_not : public unary_function<Type, Type>
{
    Type operator()(const Type& Right) const;
};

// specialized transparent functor for operator~
template <>
```

```
struct bit_not<void>
{
    template <class Type>
    auto operator()(Type&& Right) const -> decltype(~std::forward<Type>
(Right));
};
```

Parameters

Type

A type that supports a unary `operator~`.

Right

The operand of the bitwise complement operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of an lvalue or rvalue reference argument of inferred type *Type*.

Return Value

The result of `~ Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator~`.

Remarks

The `bit_not` functor is restricted to integral types for the basic data types, or to user-defined types that implement binary `operator~`.

`bit_or`

A predefined function object that does a bitwise OR operation (`operator|`) on its arguments.

C++

```
template <class Type = void>
struct bit_or : public binary_function<Type, Type, Type>
{
    Type operator()(
    const Type& Left,
    const Type& Right) const;
};

// specialized transparent functor for operator|
```

```
template <>
struct bit_or<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) | std::forward<U>(Right));
};
```

Parameters

`Type`, `T`, `U`

Any type that supports an `operator|` that takes operands of the specified or inferred types.

`Left`

The left operand of the bitwise OR operation. The unspecialized template takes an lvalue reference argument of type `Type`. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type `T`.

`Right`

The right operand of the bitwise OR operation. The unspecialized template takes an lvalue reference argument of type `Type`. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type `U`.

Return Value

The result of `Left | Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator|`.

Remarks

The `bit_or` functor is restricted to integral types for the basic data types, or to user-defined types that implement `operator|`.

`bit_xor`

A predefined function object that does a bitwise XOR operation (binary `operator^`) on its arguments.

```
C++
```

```
template <class Type = void>
struct bit_xor : public binary_function<Type, Type, Type>
{
    Type operator()(
    const Type& Left,
    const Type& Right) const;
};

// specialized transparent functor for operator^
template <>
struct bit_xor<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) ^ std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*

Any type that supports an `operator^` that takes operands of the specified or inferred types.

Left

The left operand of the bitwise XOR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right

The right operand of the bitwise XOR operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left ^ Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator^`.

Remarks

The `bit_xor` functor is restricted to integral types for the basic data types, or to user-defined types that implement binary `operator^`.

cref

Constructs a const `reference_wrapper` from an argument.

C++

```
template <class Ty>
reference_wrapper<const Ty> cref(const Ty& arg);

template <class Ty>
reference_wrapper<const Ty> cref(const reference_wrapper<Ty>& arg);
```

Parameters

Ty

The type of the argument to wrap.

arg

The argument to wrap.

Remarks

The first function returns `reference_wrapper<const Ty>(arg.get())`. You use it to wrap a const reference. The second function returns `reference_wrapper<const Ty>(arg)`. You use it to rewrap a wrapped reference as a const reference.

Example

C++

```
// std__functional__cref.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    int i = 1;

    std::cout << "i = " << i << std::endl;
```

```
    std::cout << "cref(i) = " << std::cref(i) << std::endl;
    std::cout << "cref(neg)(i) = "
        << std::cref(&neg)(i) << std::endl;

    return (0);
}
```

Output

```
i = 1
cref(i) = 1
cref(neg)(i) = -1
```

invoke

Invokes any callable object with the given arguments. Added in C++17.

C++

```
template <class Callable, class... Args>
invoke_result_t<Callable, Args...>
    invoke(Callable&& fn, Args&&... args) noexcept(/* specification */);
```

Parameters

`Callable`

The type of the object to call.

`Args`

The types of the call arguments.

`fn`

The object to call.

`args`

The call arguments.

`specification`

The **`noexcept`** specification `std::is_nothrow_invocable_v<Callable, Args>)`.

Remarks

Invokes the callable object `fn` using the parameters `args`. Effectively, `INVOKE(std::forward<Callable>(fn), std::forward<Args>(args)...)`, where the pseudo-function `INVOKE(f, t1, t2, ..., tN)` means one of the following things:

- `(t1.*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`. That is, when `std::is_base_of<T, std::decay_t<decltype(t1)>>::value` is true.
- `(t1.get().*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `std::decay_t<decltype(t1)>` is a specialization of `std::reference_wrapper`.
- `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to member function of class `T` and `t1` isn't one of the previous types.
- `t1.*f` when N == 1 and `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`. That is, when `std::is_base_of<T, std::decay_t<decltype(t1)>>::value` is true.
- `t1.get().*f` when N == 1 and `f` is a pointer to member data of a class `T` and `std::decay_t<decltype(t1)>` is a specialization of `std::reference_wrapper`.
- `(*t1).*f` when N == 1 and `f` is a pointer to member data of a class `T` and `t1` isn't one of the previous types.
- `f(t1, t2, ..., tN)` in all other cases.

For information on the result type of a callable object, see invoke_result. For predicates on callable types, see is_invocable, is_invocable_r, is_nothrow_invocable, is_nothrow_invocable_r classes.

Example

C++

```
// functional_invoke.cpp
// compile using: cl /EHsc /std:c++17 functional_invoke.cpp
#include <functional>
#include <iostream>

struct Demo
{
    int n_;
```

```
    Demo(int const n) : n_{n} {}

    void operator()( int const i, int const j ) const
    {
        std::cout << "Demo operator( " << i << ", "
            << j << " ) is " << i * j << "\n";
    }

    void difference( int const i ) const
    {
        std::cout << "Demo.difference( " << i << " ) is "
            << n_ - i << "\n";
    }
};

void divisible_by_3(int const i)
{
    std::cout << i << ( i % 3 == 0 ? " is" : " isn't" )
        << " divisible by 3.\n";
}

int main()
{
    Demo d{ 42 };
    Demo * pd{ &d };
    auto pmf = &Demo::difference;
    auto pmd = &Demo::n_;

    // Invoke a function object, like calling d( 3, -7 )
    std::invoke( d, 3, -7 );

    // Invoke a member function, like calling
    // d.difference( 29 ) or (d.*pmf)( 29 )
    std::invoke( &Demo::difference, d, 29 );
    std::invoke( pmf, pd, 13 );

    // Invoke a data member, like access to d.n_ or d.*pmd
    std::cout << "d.n_: " << std::invoke( &Demo::n_, d ) << "\n";
    std::cout << "pd->n_: " << std::invoke( pmd, pd ) << "\n";

    // Invoke a stand-alone (free) function
    std::invoke( divisible_by_3, 42 );

    // Invoke a lambda
    auto divisible_by_7 = []( int const i )
    {
        std::cout << i << ( i % 7 == 0 ? " is" : " isn't" )
            << " divisible by 7.\n";
    };
    std::invoke( divisible_by_7, 42 );
}
```

Output

```
Demo operator( 3, -7 ) is -21
Demo.difference( 29 ) is 13
Demo.difference( 13 ) is 29
d.n_: 42
pd->n_: 42
42 is divisible by 3.
42 is divisible by 7.
```

`mem_fn`

Generates a simple call wrapper.

C++

```
template <class RTy, class Ty>
unspecified mem_fn(RTy Ty::*pm);
```

Parameters

`RTy`
The return type of the wrapped function.

`Ty`
The type of the member function pointer.

Remarks

The template function returns a simple call wrapper `cw`, with a weak result type, such that the expression `cw(t, a2, ..., aN)` is the same as `INVOKE(pm, t, a2, ..., aN)`. It doesn't throw any exceptions.

The returned call wrapper is derived from `std::unary_function<cv Ty*, RTy>` (and defining the nested type `result_type` as a synonym for *`RTy`* and the nested type `argument_type` as a synonym for `cv Ty*`) only if the type *`Ty`* is a pointer to member function with cv-qualifier `cv` that takes no arguments.

The returned call wrapper is derived from `std::binary_function<cv Ty*, T2, RTy>` (and defining the nested type `result_type` as a synonym for *`RTy`*, the nested type `first argument_type` as a synonym for `cv Ty*`, and the nested type `second argument_type` as a

synonym for `T2`) only if the type *`Ty`* is a pointer to member function with cv-qualifier `cv` that takes one argument, of type `T2`.

Example

C++

```
// std__functional__mem_fn.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

class Funs
{
public:
    void square(double x)
    {
        std::cout << x << "^2 == " << x * x << std::endl;
    }

    void product(double x, double y)
    {
        std::cout << x << "*" << y << " == " << x * y << std::endl;
    }
};

int main()
{
    Funs funs;

    std::mem_fn(&Funs::square)(funs, 3.0);
    std::mem_fn(&Funs::product)(funs, 3.0, 2.0);

    return (0);
}
```

Output

```
3^2 == 9
3*2 == 6
```

`mem_fun`

Helper template functions used to construct function object adaptors for member functions when initialized with pointer arguments. Deprecated in C++11 for mem_fn and bind, and removed in C++17.

C++

```
template <class Result, class Type>
mem_fun_t<Result, Type> mem_fun (Result(Type::* pMem)());

template <class Result, class Type, class Arg>
mem_fun1_t<Result, Type, Arg> mem_fun(Result (Type::* pMem)(Arg));

template <class Result, class Type>
const_mem_fun_t<Result, Type> mem_fun(Result (Type::* pMem)() const);

template <class Result, class Type, class Arg>
const_mem_fun1_t<Result, Type, Arg> mem_fun(Result (Type::* pMem)(Arg)
const);
```

Parameters

pMem

A pointer to the member function of class `Type` to be converted to a function object.

Return Value

A `const` or non-const function object of type `mem_fun_t` or `mem_fun1_t`.

Example

C++

```
// functional_mem_fun.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class StoreVals
{
    int val;
public:
    StoreVals() { val = 0; }
    StoreVals(int j) { val = j; }

    bool display() { cout << val << " "; return true; }
    int squareval() { val *= val; return val; }
    int lessconst(int k) {val -= k; return val; }
};
```

```
int main( )
{
    vector<StoreVals *> v1;

    StoreVals sv1(5);
    v1.push_back(&sv1);
    StoreVals sv2(10);
    v1.push_back(&sv2);
    StoreVals sv3(15);
    v1.push_back(&sv3);
    StoreVals sv4(20);
    v1.push_back(&sv4);
    StoreVals sv5(25);
    v1.push_back(&sv5);

    cout << "The original values stored are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>
(&StoreVals::display));
    cout << endl;

    // Use of mem_fun calling member function through a pointer
    // square each value in the vector using squareval ()
    for_each(v1.begin(), v1.end(), mem_fun<int, StoreVals>
(&StoreVals::squareval));
    cout << "The squared values are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>
(&StoreVals::display));
    cout << endl;

    // Use of mem_fun1 calling member function through a pointer
    // subtract 5 from each value in the vector using lessconst ()
    for_each(v1.begin(), v1.end(),
        bind2nd (mem_fun1<int, StoreVals,int>(&StoreVals::lessconst), 5));
    cout << "The squared values less 5 are: " ;
    for_each(v1.begin(), v1.end(), mem_fun<bool, StoreVals>
(&StoreVals::display));
    cout << endl;
}
```

mem_fun_ref

Helper template functions used to construct function object adaptors for member functions when initialized by using reference arguments. Deprecated in C++11, removed in C++17.

C++

```
template <class Result, class Type>
mem_fun_ref_t<Result, Type> mem_fun_ref(Result (Type::* pMem)());
```

```
template <class Result, class Type, class Arg>
mem_fun1_ref_t<Result, Type, Arg> mem_fun_ref(Result (Type::* pMem)(Arg));

template <class Result, class Type>
const_mem_fun_ref_t<Result, Type> mem_fun_ref(Result Type::* pMem)() const);

template <class Result, class Type, class Arg>
const_mem_fun1_ref_t<Result, Type, Arg> mem_fun_ref(Result (T::* pMem)(Arg)
const);
```

Parameters

pMem

A pointer to the member function of class `Type` to be converted to a function object.

Return Value

A `const` or `non_const` function object of type `mem_fun_ref_t` or `mem_fun1_ref_t`.

Example

C++

```
// functional_mem_fun_ref.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

class NumVals
{
   int val;
   public:
   NumVals ( ) { val = 0; }
   NumVals ( int j ) { val = j; }

   bool display ( ) { cout << val << " "; return true; }
   bool isEven ( ) { return ( bool )  !( val %2 ); }
   bool isPrime( )
   {
      if (val < 2) { return true; }
      for (int i = 2; i <= val / i; ++i)
      {
         if (val % i == 0) { return false; }
      }
```

```
      return true;
   }
};

int main( )
{
   vector <NumVals> v1 ( 13 ), v2 ( 13 );
   vector <NumVals>::iterator v1_Iter, v2_Iter;
   int i, k;

   for ( i = 0; i < 13; i++ ) v1 [ i ] = NumVals ( i+1 );
   for ( k = 0; k < 13; k++ ) v2 [ k ] = NumVals ( k+1 );

   cout << "The original values stored in v1 are: " ;
   for_each( v1.begin( ), v1.end( ),
   mem_fun_ref ( &NumVals::display ) );
   cout << endl;

   // Use of mem_fun_ref calling member function through a reference
   // remove the primes in the vector using isPrime ( )
   v1_Iter = remove_if ( v1.begin( ),  v1.end( ),
      mem_fun_ref ( &NumVals::isPrime ) );
   cout << "With the primes removed, the remaining values in v1 are: " ;
   for_each( v1.begin( ), v1_Iter,
   mem_fun_ref ( &NumVals::display ) );
   cout << endl;

   cout << "The original values stored in v2 are: " ;
   for_each( v2.begin( ), v2.end( ),
   mem_fun_ref ( &NumVals::display ) );
   cout << endl;

   // Use of mem_fun_ref calling member function through a reference
   // remove the even numbers in the vector v2 using isEven ( )
   v2_Iter = remove_if ( v2.begin( ),  v2.end( ),
      mem_fun_ref ( &NumVals::isEven ) );
   cout << "With the even numbers removed, the remaining values are: " ;
   for_each( v2.begin( ),  v2_Iter,
   mem_fun_ref ( &NumVals::display ) );
   cout << endl;
}
```

Output

```
The original values stored in v1 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the primes removed, the remaining values in v1 are: 4 6 8 9 10 12
The original values stored in v2 are: 1 2 3 4 5 6 7 8 9 10 11 12 13
With the even numbers removed, the remaining values are: 1 3 5 7 9 11 13
```

not1

Returns the complement of a unary predicate. Deprecated for not_fn in C++17.

```
C++
```

```
template <class UnaryPredicate>
unary_negate<UnaryPredicate> not1(const UnaryPredicate& predicate);
```

Parameters

predicate

The unary predicate to be negated.

Return Value

A unary predicate that is the negation of the unary predicate modified.

Remarks

If a `unary_negate` is constructed from a unary predicate `predicate(x)`, then it returns `!predicate(x)`.

Example

```
C++
```

```
// functional_not1.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
```

```
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        not1(bind2nd(greater<int>(), 10)));

    cout << "The number of elements in v1 not greater than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.
```

not2

Returns the complement of a binary predicate. Deprecated for not_fn in C++17.

C++

```
template <class BinaryPredicate>
binary_negate<BinaryPredicate> not2(const BinaryPredicate& func);
```

Parameters

func

The binary predicate to be negated.

Return Value

A binary predicate that is the negation of the binary predicate modified.

Remarks

If a `binary_negate` is constructed from a binary predicate `binary_predicate(x, y)`, then it returns `!binary_predicate(x, y)`.

Example

C++

```
// functional_not2.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   for ( i = 0 ; i < 5 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order,
   // use the binary_negate helper function not2
   sort( v1.begin( ), v1.end( ), not2(less<int>( ) ) );
   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 6262 6262 41 18467 6334 26500 19169 )
Sorted vector v1 = ( 41 6262 6262 6334 18467 19169 26500 )
Resorted vector v1 = ( 26500 19169 18467 6334 6262 6262 41 )
```

not_fn

The `not_fn` function template takes a callable object and returns a callable object. When the returned callable object is later invoked with some arguments, it passes them to the original callable object, and logically negates the result. It preserves the const qualification and value category behavior of the wrapped callable object. `not_fn` is new in C++17, and replaces the deprecated `std::not1`, `std::not2`, `std::unary_negate`, and `std::binary_negate`.

C++

```
template <class Callable>
/* unspecified */ not_fn(Callable&& func);
```

Parameters

`func`
A callable object used to construct the forwarding call wrapper.

Remarks

The template function returns a call wrapper like `return call_wrapper(std::forward<Callable>(func))`, based on this exposition-only class:

C++

```
class call_wrapper
{
   using FD = decay_t<Callable>;
   explicit call_wrapper(Callable&& func);

public:
   call_wrapper(call_wrapper&&) = default;
   call_wrapper(call_wrapper const&) = default;

   template<class... Args>
     auto operator()(Args&&...) & -> decltype(!declval<invoke_result_t<FD&
(Args...)>>());
```

```
    template<class... Args>
      auto operator()(Args&&...) const& ->
decltype(!declval<invoke_result_t<FD const&(Args...)>>());

    template<class... Args>
      auto operator()(Args&&...) && ->
decltype(!declval<invoke_result_t<FD(Args...)>>());

    template<class... Args>
      auto operator()(Args&&...) const&& ->
decltype(!declval<invoke_result_t<FD const(Args...)>>());

private:
  FD fd;
};
```

The explicit constructor on the callable object *`func`* requires type `std::decay_t<Callable>` to satisfy the requirements of `MoveConstructible`, and `is_constructible_v<FD, Callable>` must be true. It initializes the wrapped callable object `fd` from `std::forward<Callable>(func)`, and throws any exception thrown by construction of `fd`.

The wrapper exposes call operators distinguished by lvalue or rvalue reference category and const qualification as shown here:

C++

```
template<class... Args> auto operator()(Args&&... args) & ->
decltype(!declval<invoke_result_t<FD&(Args...)>>());
template<class... Args> auto operator()(Args&&... args) const& ->
decltype(!declval<invoke_result_t<FD const&(Args...)>>());
template<class... Args> auto operator()(Args&&... args) && ->
decltype(!declval<invoke_result_t<FD(Args...)>>());
template<class... Args> auto operator()(Args&&... args) const&& ->
decltype(!declval<invoke_result_t<FD const(Args...)>>());
```

The first two are the same as `return !std::invoke(fd, std::forward<Args>(args)...)`. The second two are the same as `return !std::invoke(std::move(fd), std::forward<Args>(args)...)`.

Example

C++

```
// functional_not_fn_.cpp
// compile with: /EHsc /std:c++17
```

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main()
{
    std::vector<int> v1 = { 99, 6264, 41, 18467, 6334, 26500, 19169 };
    auto divisible_by_3 = [](int i){ return i % 3 == 0; };

    std::cout << "Vector v1 = ( " ;
    for (const auto& item : v1)
    {
        std::cout << item << " ";
    }
    std::cout << ")" << std::endl;

    // Count the number of vector elements divisible by 3.
    int divisible =
        std::count_if(v1.begin(), v1.end(), divisible_by_3);
    std::cout << "Elements divisible by three: "
        << divisible << std::endl;

    // Count the number of vector elements not divisible by 3.
    int not_divisible =
        std::count_if(v1.begin(), v1.end(), std::not_fn(divisible_by_3));
    std::cout << "Elements not divisible by three: "
        << not_divisible << std::endl;
}
```

Output

```
Vector v1 = ( 99 6264 41 18467 6334 26500 19169 )
Elements divisible by three: 2
Elements not divisible by three: 5
```

ptr_fun

Helper template functions used to convert unary and binary function pointers, respectively, into unary and binary adaptable functions. Deprecated in C++11, removed in C++17.

C++

```
template <class Arg, class Result>
pointer_to_unary_function<Arg, Result, Result (*)(Arg)> ptr_fun(Result
(*pfunc)(Arg));

template <class Arg1, class Arg2, class Result>
```

```
pointer_to_binary_function<Arg1, Arg2, Result, Result (*)(Arg1, Arg2)>
ptr_fun(Result (*pfunc)(Arg1, Arg2));
```

Parameters

pfunc

The unary or binary function pointer to be converted to an adaptable function.

Return Value

The first template function returns the unary function pointer_to_unary_function <`Arg`, **`Result`**>(* `pfunc`).

The second template function returns binary function pointer_to_binary_function <**`Arg1`**, **`Arg2`**, **`Result`**>(* `pfunc`).

Remarks

A function pointer is a function object. It may be passed to any algorithm that expects a function as a parameter, but it isn't adaptable. Information about its nested types is required to use it with an adaptor, for example, to bind a value to it or to negate it. The conversion of unary and binary function pointers by the `ptr_fun` helper function allows the function adaptors to work with unary and binary function pointers.

Example

C++

```
// functional_ptr_fun.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
#include <iostream>

int main( )
{
    using namespace std;
    vector <char*> v1;
    vector <char*>::iterator Iter1, RIter;

    v1.push_back ( "Open" );
    v1.push_back ( "up" );
```

```
    v1.push_back ( "the" );
    v1.push_back ( "opalescent" );
    v1.push_back ( "gates" );

    cout << "Original sequence contains: " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; ++Iter1 )
        cout << *Iter1 << " ";
    cout << endl;

    // To search the sequence for "opalescent"
    // use a pointer_to_function conversion
    RIter = find_if( v1.begin( ), v1.end( ),
        not1 ( bind2nd (ptr_fun ( strcmp ), "opalescent" ) ) );

    if ( RIter != v1.end( ) )
    {
        cout << "Found a match: "
            << *RIter << endl;
    }
}
```

ref

Constructs a `reference_wrapper` from an argument.

C++

```
template <class Ty>
    reference_wrapper<Ty> ref(Ty& arg);

template <class Ty>
    reference_wrapper<Ty> ref(reference_wrapper<Ty>& arg);
```

Return Value

A reference to `arg`; specifically, `reference_wrapper<Ty>(arg)`.

Example

The following example defines two functions: one bound to a string variable, the other bound to a reference of the string variable computed by a call to `ref`. When the value of the variable changes, the first function continues to use the old value and the second function uses the new value.

C++

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <ostream>
#include <string>
#include <vector>
using namespace std;
using namespace std;
using namespace std::placeholders;

bool shorter_than(const string& l, const string& r)
{
    return l.size() < r.size();
}

int main()
{
    vector<string> v_original;
    v_original.push_back("tiger");
    v_original.push_back("cat");
    v_original.push_back("lion");
    v_original.push_back("cougar");

    copy(v_original.begin(), v_original.end(), ostream_iterator<string>
(cout, " "));
    cout << endl;

    string s("meow");

    function<bool (const string&)> f = bind(shorter_than, _1, s);
    function<bool (const string&)> f_ref = bind(shorter_than, _1, ref(s));

    vector<string> v;

    // Remove elements that are shorter than s ("meow")

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f), v.end());

    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;

    // Now change the value of s.
    // f_ref, which is bound to ref(s), will use the
    // new value, while f is still bound to the old value.

    s = "kitty";

    // Remove elements that are shorter than "meow" (f is bound to old value
of s)

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f), v.end());
```

```
    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;

    // Remove elements that are shorter than "kitty" (f_ref is bound to
ref(s))

    v = v_original;
    v.erase(remove_if(v.begin(), v.end(), f_ref), v.end());

    copy(v.begin(), v.end(), ostream_iterator<string>(cout, " "));
    cout << endl;
}
```

Output

```
tiger cat lion cougar
tiger lion cougar
tiger lion cougar
tiger cougar
```

swap

Swaps two `function` objects.

C++

```
template <class FT>
    void swap(function<FT>& f1, function<FT>& f2);
```

Parameters

`FT`

The type controlled by the function objects.

`f1`

The first function object.

`f2`

The second function object.

Remarks

The function returns `f1.swap(f2)`.

Example

C++

```cpp
// std__functional__swap.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
{
    return (-val);
}

int main()
{
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << std::endl;

    swap(fn0, fn1);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
}
```

Output

```
empty == false
val == -3
empty == true

empty == true
empty == false
val == -3
```

`<functional>` operators

Article • 12/07/2021

operator==

Tests if callable object is empty.

C++

```
template <class Fty>
    bool operator==(const function<Fty>& f, null_ptr_type npc);

template <class Fty>
    bool operator==(null_ptr_type npc, const function<Fty>& f);
```

Parameters

Fty
The function type to wrap.

f
The function object

npc
A null pointer.

Remarks

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer constant. Both return true only if the `function` object is empty.

Example

C++

```
// std__functional__operator_eq.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
```

```
{
    return (-val);
}

int main()
{
    std::function<int(int)> fn0;
    std::cout << std::boolalpha << "empty == "
        << (fn0 == 0) << std::endl;

    std::function<int(int)> fn1(neg);
    std::cout << std::boolalpha << "empty == "
        << (fn1 == 0) << std::endl;

    return (0);
}
```

Output

```
empty == true
empty == false
```

operator!=

Tests if callable object is not empty.

C++

```
template <class Fty>
    bool operator!=(const function<Fty>& f, null_ptr_type npc);

template <class Fty>
    bool operator!=(null_ptr_type npc, const function<Fty>& f);
```

Parameters

Fty
The function type to wrap.

f
The function object

npc
A null pointer.

Remarks

The operators both take an argument that is a reference to a `function` object and an argument that is a null pointer constant. Both return true only if the `function` object is not empty.

Example

C++

```
// std__functional__operator_ne.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn0;
    std::cout << std::boolalpha << "not empty == "
        << (fn0 != 0) << std::endl;

    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "not empty == "
        << (fn1 != 0) << std::endl;

    return (0);
    }
```

Output

```
not empty == false
not empty == true
```

`_1` Object

Article • 08/03/2021

Placeholders for replaceable arguments.

Syntax

C++

```
namespace placeholders {
    extern unspecified _1, _2, ... _N
} // namespace placeholders (within std)
```

Remarks

The objects `_1, _2, ... _N` are placeholders that represent the first, second, ..., through Nth argument, respectively in a function call to an object returned by bind. For example, you use `_6` to specify where the sixth argument should be inserted when the `bind` expression is evaluated.

In the Microsoft implementation, the value of `_N` is 20.

Example

C++

```
// std__functional_placeholder.cpp
// compile with: /EHsc
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std::placeholders;

void square(double x)
    {
    std::cout << x << "^2 == " << x * x << std::endl;
    }

void product(double x, double y)
    {
    std::cout << x << "*" << y << " == " << x * y << std::endl;
    }
```

```
int main()
    {
    double arg[] = {1, 2, 3};

    std::for_each(&arg[0], &arg[3], square);
    std::cout << std::endl;

    std::for_each(&arg[0], &arg[3], std::bind(product, _1, 2));
    std::cout << std::endl;

    std::for_each(&arg[0], &arg[3], std::bind(square, _1));

    return (0);
    }
```

Output

```
1^2 == 1
2^2 == 4
3^2 == 9

1*2 == 2
2*2 == 4
3*2 == 6

1^2 == 1
2^2 == 4
3^2 == 9
```

bad_function_call Class

Article • 08/03/2021

Reports a bad function call.

Syntax

```
C++
```

```
class bad_function_call : public std::exception {};
```

Remarks

The class describes an exception thrown to indicate that a call to `operator()` on a function Class

binary_function Struct

Article • 08/03/2021

An empty base struct that defines types that may be inherited by derived classes that provides a binary function object. Deprecated in C++11, removed in C++17.

Syntax

C++

```
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

Remarks

The template struct serves as a base for classes that define a member function of the form:

> *result_type* ** operator()(const** *first_argument_type*&, **const** *second_argument_type*&) **const**

All such binary functions can refer to their first argument type as *first_argument_type*, their second argument type as *second_argument_type*, and their return type as *result_type*.

Example

C++

```
// functional_binary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

template <class Type> class average:
```

```
binary_function<Type, Type, Type>
{
public:
   result_type operator( ) ( first_argument_type a,
                  second_argument_type b )
   {
      return (result_type) ( ( a + b ) / 2 );
   }
};

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   for ( int i = 1 ; i <= 6 ; i++ )
      v1.push_back( 11.0 / i );

   for ( int j = 0 ; j <= 5 ; j++ )
      v2.push_back( -2.0 * j );

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise averages of the elements of v1 & v2
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
      average<double>( ) );

   cout << "The element-wise averages are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 11 5.5 3.66667 2.75 2.2 1.83333 )
The vector v2 = ( -0 -2 -4 -6 -8 -10 )
The element-wise averages are: ( 5.5 1.75 -0.166667 -1.625 -2.9 -4.08333 )
```

binary_negate Class

Article • 08/03/2021

A class template providing a member function that negates the return value of a specified binary function. Deprecated in C++17 in favor of not_fn.

Syntax

```
C++
```

```
template <class Operation>
class binary_negate
    : public binaryFunction <typename Operation::first_argument_type,
                             typename Operation::second_argument_type,
bool>
{
    explicit binary_negate(const Operation& Func);
    bool operator()(const typename Operation::first_argument_type& left,
                    const typename Operation::second_argument_type& right)
const;
};
```

Parameters

Func
The binary function to be negated.

left
The left operand of the binary function to be negated.

right
The right operand of the binary function to be negated.

Return Value

The negation of the binary function.

Remarks

The class template stores a copy of a binary function object *Func*. It defines its member function `operator()` as returning `!Func(left, right)`.

The constructor of `binary_negate` is rarely used directly. The helper function not2 is usually preferred to declare and use the **binary_negator** adaptor predicate.

Example

C++

```
// functional_binary_negate.cpp
// compile with: /EHsc
#define _CRT_RAND_S
#include <stdlib.h>

#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   vector <unsigned int> v1;
   vector <unsigned int>::iterator Iter1;

   unsigned int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   unsigned int randVal = 0;
   for ( i = 0 ; i < 5 ; i++ )
   {
      rand_s(&randVal);
      v1.push_back( randVal );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<unsigned int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in descending order,
   // use the binary_negate function
   sort( v1.begin( ), v1.end( ),
   binary_negate<less<unsigned int> >(less<unsigned int>( ) ) );

   // The helper function not2 could also have been used
```

```
   // in the above line and is usually preferred for convenience
   // sort( v1.begin( ), v1.end( ), not2(less<unsigned int>( ) ) );

   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = ( 6262 6262 2233879413 2621500314 580942933 3715465425
3739828298 )
Sorted vector v1 = ( 6262 6262 580942933 2233879413 2621500314 3715465425
3739828298 )
Resorted vector v1 = ( 3739828298 3715465425 2621500314 2233879413 580942933
6262 6262 )
```

binder1st Class

Article • 08/03/2021

A class template providing a constructor that converts a binary function object into a unary function object by binding the first argument of the binary function to a specified value. Deprecated in C++11 in favor of bind, and removed in C++17.

Syntax

C++

```
template <class Operation>
class binder1st
    : public unaryFunction <typename Operation::second_argument_type,
                            typename Operation::result_type>
{
public:
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;
    binder1st(
        const Operation& binary_fn,
        const typename Operation::first_argument_type& left);

    result_type operator()(const argument_type& right) const;
    result_type operator()(const argument_type& right) const;

protected:
    Operation op;
    typename Operation::first_argument_type value;
};
```

Parameters

binary_fn
The binary function object to be converted to a unary function object.

left
The value to which the first argument of the binary function object is to be bound.

right
The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return Value

The unary function object that results from binding the first argument of the binary function object to the value *left*.

Remarks

The class template stores a copy of a binary function object *binary_fn* in `op`, and a copy of *left* in `value`. It defines its member function `operator()` as returning `op(value, right)`.

If *binary_fn* is an object of type `Operation` and `c` is a constant, then `bind1st(binary_fn, c)` is a more convenient equivalent to `binder1st<Operation>(binary_fn, c)`. For more information, see bind1st.

Example

C++

```
// functional_binder1st.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
```

```
        binder1st<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    // Compare use of binder2nd fixing 2nd argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder2nd<less<int> >(less<int>(), 10));
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
```

binder2nd Class

Article • 08/03/2021

A class template providing a constructor that converts a binary function object into a unary function object by binding the second argument of the binary function to a specified value. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Operation>
class binder2nd
    : public unaryFunction <typename Operation::first_argument_type,
    typename Operation::result_type>
{
    typedef typename Operation::argument_type argument_type;
    typedef typename Operation::result_type result_type;
    binder2nd(
        const Operation& func,
        const typename Operation::second_argument_type& right);

    result_type operator()(const argument_type& left) const;
    result_type operator()(argument_type& left) const;
};
```

Parameters

func
The binary function object to be converted to a unary function object.

right
The value to which the second argument of the binary function object is to be bound.

left
The value of the argument that the adapted binary object compares to the fixed value of the second argument.

Return value

The unary function object that results from binding the second argument of the binary function object to the value *right*.

Remarks

The class template stores a copy of a binary function object *func* in `op`, and a copy of *right* in `value`. It defines its member function `operator()` as returning `op(left, value)`.

If *func* is an object of type `Operation` and c is a constant, then bind2nd`(func, c)` is equivalent to the `binder2nd` class constructor `binder2nd<Operation>(func, c)`, and more convenient.

Example

C++

```
// functional_binder2nd.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    // Count the number of integers > 10 in the vector
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(),
        binder2nd<greater<int> >(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    // Compare using binder1st fixing 1st argument:
    // count the number of integers < 10 in the vector
    vector<int>::iterator::difference_type result2;
    result2 = count_if(v1.begin(), v1.end(),
        binder1st<greater<int> >(greater<int>(), 10));
```

```
    cout << "The number of elements in v1 less than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
The number of elements in v1 less than 10 is: 2.
```

const_mem_fun_ref_t Class

Article • 08/03/2021

An adapter class that allows a `const` member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type>
    class const_mem_fun_ref_t
: public unary_function<Type, Result>
{
    explicit const_mem_fun_t(Result (Type::* Pm)() const);
    Result operator()(const Type& left) const;
};
```

Parameters

Pm
A pointer to the member function of class `Type` to be converted to a function object.

left
The object that the *Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The class template stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (**left**.* `Pm`)() `const`.

Example

The constructor of `const_mem_fun_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

const_mem_fun_t Class

Article • 08/03/2021

An adapter class that allows a const member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type>
class const_mem_fun_t : public unary_function <Type *, Result>
{
    explicit const_mem_fun_t(Result (Type::* Pm)() const);
    Result operator()(const Type* Pleft) const;
};
```

Parameters

Pm
A pointer to the member function of class `Type` to be converted to a function object.

Pleft
The object that the *Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The class template stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (`Pleft->* Pm`)() **`const`**.

Example

The constructor of `const_mem_fun_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

const_mem_fun1_ref_t Class

Article • 08/03/2021

An adapter class that allows a `const` member function that takes a single argument to be called as a binary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type, class Arg>
    class const_mem_fun1_ref_t
        : public binary_function<Type, Arg, Result>
{
    explicit const_mem_fun1_ref_t(Result (Type::* Pm)(Arg) const);
    Result operator()(const Type& left, Arg right) const;
};
```

Parameters

Pm
A pointer to the member function of class `Type` to be converted to a function object.

left
The `const` object that the *Pm* member function is called on.

right
The argument that is being given to *Pm*.

Return Value

An adaptable binary function.

Remarks

The class template stores a copy of *Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (`left`.* *Pm*)(`right`) `const`.

Example

The constructor of `const_mem_fun1_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See mem_fun_ref for examples of how to use member function adaptors.

const_mem_fun1_t Class

Article • 08/03/2021

An adapter class that allows a `const` member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type, class Arg>
class const_mem_fun1_t : public binary_function<const Type *, Arg, Result>
{
    explicit const_mem_fun1_t(Result (Type::* member_ptr)(Arg) const);
    Result operator()(const Type* left, Arg right) const;
};
```

Parameters

member_ptr
A pointer to the member function of class `Type` to be converted to a function object.

left
The `const` object that the *member_ptr* member function is called on.

right
The argument that is being given to *member_ptr*.

Return Value

An adaptable binary function.

Remarks

The class template stores a copy of *member_ptr*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning `(left->member_ptr)(right) const`.

Example

The constructor of `const_mem_fun1_t` is rarely used directly. `mem_fn` is used to adapt member functions. See mem_fn for an example of how to use member function adaptors.

divides Struct

Article • 08/03/2021

A predefined function object that performs the division operation (`operator/`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct divides : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator/
template <>
struct divides<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const
    -> decltype(std::forward<T>(Left)*/ std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
A type that supports an `operator/` that takes operands of the specified or inferred types.

Left
The left operand of the division operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the division operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left / Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator/`.

Example

C++

```
// functional_divides.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 (6);
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 7.0 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 2.0 * j);
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise quotients of the elements of v1 & v2
   transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
      divides<double>( ) );

   cout << "The element-wise quotients are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 0 7 14 21 28 35 )
The vector v2 = ( 2 4 6 8 10 12 )
The element-wise quotients are: ( 0 1.75 2.33333 2.625 2.8 2.91667 )
```

equal_to Struct

Article • 08/03/2021

A binary predicate that performs the equality operation (`operator==`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct equal_to : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator==
template <>
struct equal_to<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
      ->  decltype(std::forward<T>(Left) == std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator==` that takes operands of the specified or inferred types.

Left
The left operand of the equality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the equality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left == Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator==`.

Remarks

The objects of type *Type* must be equality-comparable. This requires that the `operator==` defined on the set of objects satisfies the mathematical properties of an equivalence relation. All of the built-in numeric and pointer types satisfy this requirement.

Example

C++

```
// functional_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i+=2 )
   {
      v1.push_back( 2.0 *i );
      v1.push_back( 2.0 * i + 1.0 );
   }

   int j;
   for ( j = 0 ; j <= 5 ; j+=2 )
   {
      v2.push_back( - 2.0 * j );
      v2.push_back( 2.0 * j + 1.0 );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
```

```
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Testing for the element-wise equality between v1 & v2
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
      equal_to<double>( ) );

   cout << "The result of the element-wise equal_to comparison\n"
      << "between v1 & v2 is: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise equal_to comparison
between v1 & v2 is: ( 1 1 0 1 0 1 )
```

function Class

Article • 06/17/2022

Wrapper for a callable object.

Syntax

C++

```
template <class Fty>
class function  // Fty of type Ret(T1, T2, ..., TN)
    : public unary_function<T1, Ret>       // when Fty is Ret(T1)
    : public binary_function<T1, T2, Ret>  // when Fty is Ret(T1, T2)
{
public:
    typedef Ret result_type;

    function();
    function(nullptr_t);
    function(const function& right);
    template <class Fty2>
        function(Fty2 fn);
    template <class Fty2, class Alloc>
        function(reference_wrapper<Fty2>, const Alloc& Ax);

    template <class Fty2, class Alloc>
        void assign(Fty2, const Alloc& Ax);
    template <class Fty2, class Alloc>
        void assign(reference_wrapper<Fty2>, const Alloc& Ax);
    function& operator=(nullptr_t);
    function& operator=(const function&);
    template <class Fty2>
        function& operator=(Fty2);
    template <class Fty2>
        function& operator=(reference_wrapper<Fty2>);

    void swap(function&);
    explicit operator bool() const;

    result_type operator()(T1, T2, ....., TN) const;
    const std::type_info& target_type() const;
    template <class Fty2>
        Fty2 *target();

    template <class Fty2>
        const Fty2 *target() const;

    template <class Fty2>
        void operator==(const Fty2&) const = delete;
    template <class Fty2>
```

```
        void operator!=(const Fty2&) const = delete;
};
```

Parameters

Fty
The function type to wrap.

Ax
The allocator function.

Remarks

The class template is a call wrapper whose call signature is `Ret(T1, T2, ..., TN)`. You use it to enclose various callable objects in a uniform wrapper.

Some member functions take an operand that names the desired target object. You can specify such an operand in several ways:

`fn`: The callable object `fn`; after the call the `function` object holds a copy of `fn`

`fnref`: The callable object named by `fnref.get()`; after the call the `function` object holds a reference to `fnref.get()`

`right`: The callable object, if any, held by the `function` object `right`

`npc`: A null pointer; after the call the `function` object is empty

In all cases, `INVOKE(f, t1, t2, ..., tN)`, where `f` is the callable object and `t1, t2, ..., tN` are lvalues of types `T1, T2, ..., TN` respectively, must be well-formed and, if `Ret` isn't void, convertible to `Ret`.

An empty `function` object doesn't hold a callable object or a reference to a callable object.

Members

Constructors

Name	Description

Name	Description
function	Constructs a wrapper that either is empty or stores a callable object of arbitrary type with a fixed signature.

Typedefs

Name	Description
result_type	The return type of the stored callable object.

Functions

Name	Description
assign	Assigns a callable object to this function object.
swap	Swap two callable objects.
target	Tests if stored callable object is callable as specified.
target_type	Gets type information on the callable object.

Operators

Name	Description
operator unspecified	Tests if stored callable object exists.
operator()	Calls a callable object.
operator=	Replaces the stored callable object.

assign

Assigns a callable object to this function object.

C++

```
template <class Fx, class Alloc>
    void assign(
        Fx _Func,
        const Alloc& Ax);

template <class Fx, class Alloc>
```

```
    void assign(
        reference_wrapper<Fx> _Fnref,
        const Alloc& Ax);
```

Parameters

_Func
A callable object.

_Fnref
A reference wrapper that contains a callable object.

Ax
An allocator object.

Remarks

The member functions each replace the `callable object` held by `*this` with the callable object passed as the `operand`. Both allocate storage with the allocator object *Ax*.

function

Constructs a wrapper that either is empty or stores a callable object of arbitrary type with a fixed signature.

C++

```
function();
function(nullptr_t npc);
function(const function& right);
template <class Fx>
    function(Fx _Func);
template <class Fx>
    function(reference_wrapper<Fx> _Fnref);
template <class Fx, class Alloc>
    function(
        Fx _Func,
        const Alloc& Ax);

template <class Fx, class Alloc>
    function(
        reference_wrapper<Fx> _Fnref,
        const Alloc& Ax);
```

Parameters

right
The function object to copy.

Fx
The type of the callable object.

_Func
The callable object to wrap.

Alloc
The allocator type.

Ax
The allocator.

_Fnref
The callable object reference to wrap.

Remarks

The first two constructors construct an empty `function` object. The next three constructors construct a `function` object that holds the callable object passed as the operand. The last two constructors allocate storage with the allocator object Ax.

Example

C++

```
// std__functional__function_function.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>
#include <vector>

int square(int val)
{
    return val * val;
}

class multiply_by
{
public:
    explicit multiply_by(const int n) : m_n(n) { }

    int operator()(const int x) const
```

```
    {
        return m_n * x;
    }

private:
    int m_n;
};

int main()
{

    typedef std::vector< std::function<int (int)> > vf_t;

    vf_t v;
    v.push_back(square);
    v.push_back(std::negate<int>());
    v.push_back(multiply_by(3));

    for (vf_t::const_iterator i = v.begin(); i != v.end(); ++i)
    {
        std::cout << (*i)(10) << std::endl;
    }

    std::function<int (int)> f = v[0];
    std::function<int (int)> g;

    if (f) {
        std::cout << "f is non-empty (correct)." << std::endl;
    } else {
        std::cout << "f is empty (can't happen)." << std::endl;
    }

    if (g) {
        std::cout << "g is non-empty (can't happen)." << std::endl;
    } else {
        std::cout << "g is empty (correct)." << std::endl;
    }

    return 0;
}
```

Output

```
100
-10
30
f is non-empty (correct).
g is empty (correct).
```

operator unspecified

Tests if stored callable object exists.

C++

```
operator unspecified();
```

Remarks

The operator returns a value that is convertible to `bool` with a true value only if the object isn't empty. You use it to test whether the object is empty.

Example

C++

```
// std__functional__function_operator_bool.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn0;
    std::cout << std::boolalpha << "not empty == " << (bool)fn0 <<
std::endl;

    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "not empty == " << (bool)fn1 <<
std::endl;

    return (0);
    }
```

Output

```
not empty == false
not empty == true
```

operator()

Calls a callable object.

C++

```
result_type operator()(
    T1 t1,
    T2 t2, ...,
    TN tN);
```

Parameters

TN
The type of the Nth call argument.

tN
The Nth call argument.

Remarks

The member function returns `INVOKE(fn, t1, t2, ..., tN, Ret)`, where `fn` is the target object stored in `*this`. You use it to call the wrapped callable object.

Example

C++

```
// std__functional__function_operator_call.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
    }
```

Output

```
empty == false
val == -3
```

operator=

Replaces the stored callable object.

C++

```
function& operator=(null_ptr_type npc);
function& operator=(const function& right);
template <class Fty>
    function& operator=(Fty fn);
template <class Fty>
    function& operator=(reference_wrapper<Fty> fnref);
```

Parameters

npc
A null pointer constant.

right
The function object to copy.

fn
The callable object to wrap.

fnref
The callable object reference to wrap.

Remarks

The operators each replace the callable object held by `*this` with the callable object passed as the operand.

Example

C++

```
// std__functional__function_operator_as.cpp
// compile with: /EHsc
#include <functional>
```

```
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    fn1 = 0;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;

    fn1 = neg;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    fn1 = fn0;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    fn1 = std::cref(fn1);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
    }
```

Output

```
empty == false
val == -3
empty == true
empty == false
val == -3
empty == false
val == -3
empty == false
val == -3
```

result_type

The return type of the stored callable object.

C++

```
typedef Ret result_type;
```

Remarks

The typedef is a synonym for the type `Ret` in the template's call signature. You use it to determine the return type of the wrapped callable object.

Example

C++

```
// std__functional__function_result_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn1(neg);
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;

    std::function<int (int)>::result_type val = fn1(3);
    std::cout << "val == " << val << std::endl;

    return (0);
    }
```

Output

```
empty == false
val == -3
```

swap

Swap two callable objects.

C++

```
void swap(function& right);
```

Parameters

right
The function object to swap with.

Remarks

The member function swaps the target objects between `*this` and *right*. It does so in constant time and throws no exceptions.

Example

C++

```
// std__functional__function_swap.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "val == " << fn0(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << std::endl;

    fn0.swap(fn1);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "val == " << fn1(3) << std::endl;

    return (0);
    }
```

Output

```
empty == false
val == -3
empty == true
```

```
empty == true
empty == false
val == -3
```

target

Tests if stored callable object is callable as specified.

C++

```
template <class Fty2>
    Fty2 *target();
template <class Fty2>
    const Fty2 *target() const;
```

Parameters

Fty2
The target callable object type to test.

Remarks

The type *Fty2* must be callable for the argument types `T1, T2, ..., TN` and the return type `Ret`. If `target_type() == typeid(Fty2)`, the member template function returns the address of the target object; otherwise, it returns 0.

A type *Fty2* is callable for the argument types `T1, T2, ..., TN` and the return type `Ret` if, for lvalues `fn, t1, t2, ..., tN` of types `Fty2, T1, T2, ..., TN`, respectively, `INVOKE(fn, t1, t2, ..., tN)` is well-formed and, if `Ret` isn't **`void`**, convertible to `Ret`.

Example

C++

```
// std__functional__function_target.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }
```

```
int main()
    {
    typedef int (*Myfun)(int);
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "no target == " << (fn0.target<Myfun>() == 0) << std::endl;

    Myfun *fptr = fn0.target<Myfun>();
    std::cout << "val == " << (*fptr)(3) << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "no target == " << (fn1.target<Myfun>() == 0) << std::endl;

    return (0);
    }
```

Output

```
empty == false
no target == false
val == -3
empty == true
no target == true
```

target_type

Gets type information on the callable object.

C++

```
const std::type_info& target_type() const;
```

Remarks

The member function returns `typeid(void)` if `*this` is empty, otherwise it returns `typeid(T)`, where `T` is the type of the target object.

Example

C++

```
// std__functional__function_target_type.cpp
// compile with: /EHsc
```

```
#include <functional>
#include <iostream>

int neg(int val)
    {
    return (-val);
    }

int main()
    {
    std::function<int (int)> fn0(neg);
    std::cout << std::boolalpha << "empty == " << !fn0 << std::endl;
    std::cout << "type == " << fn0.target_type().name() << std::endl;

    std::function<int (int)> fn1;
    std::cout << std::boolalpha << "empty == " << !fn1 << std::endl;
    std::cout << "type == " << fn1.target_type().name() << std::endl;

    return (0);
    }
```

Output

```
empty == false
type == int (__cdecl*)(int)
empty == true
type == void
```

greater Struct

Article • 08/03/2021

A binary predicate that performs the greater-than operation (`operator>`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct greater : public binary_function <Type, Type, bool>
{
    bool operator()(
    const Type& Left,
    const Type& Right) const;

};

// specialized transparent functor for operator>
template <>
struct greater<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const
    ->  decltype(std::forward<T>(Left)> std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator>` that takes operands of the specified or inferred types.

Left
The left operand of the greater-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the greater-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left > Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator>`.

Remarks

The binary predicate `greater<Type>` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

C++

```
// functional_greater.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   for ( i = 0 ; i < 8 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
```

```
   cout << ")" << endl;

   // To sort in descending order,
   // specify binary predicate greater<int>( )
   sort( v1.begin( ), v1.end( ), greater<int>( ) );
   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = (41 18467 6334 26500 19169 15724 11478 29358)
Sorted vector v1 = (41 6334 11478 15724 18467 19169 26500 29358)
Resorted vector v1 = (29358 26500 19169 18467 15724 11478 6334 41)
```

greater_equal Struct

Article • 08/03/2021

A binary predicate that performs the greater-than-or-equal-to operation (`operator>=`) on its arguments.

Syntax

```
C++
```

```
template <class Type = void>
struct greater_equal : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator>=
template <>
struct greater_equal<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left)>= std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator>=` that takes operands of the specified or inferred types.

Left
The left operand of the greater-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the greater-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left >= Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator>=`.

Remarks

The binary predicate `greater_equal<Type>` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

C++

```
// functional_greater_equal.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;

   int i;
   v1.push_back( 6262 );
   v1.push_back( 6262 );
   for ( i = 0 ; i < 5 ; i++ )
   {
      v1.push_back( rand( ) );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use default binary predicate less<int>( )
   sort( v1.begin( ), v1.end( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
```

```
   cout << ")" << endl;

   // To sort in descending order,
   // specify binary predicate greater_equal<int>( )
   sort( v1.begin( ), v1.end( ), greater_equal<int>( ) );
   cout << "Resorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = (6262 6262 41 18467 6334 26500 19169)
Sorted vector v1 = (41 6262 6262 6334 18467 19169 26500)
Resorted vector v1 = (26500 19169 18467 6334 6262 6262 41)
```

hash Class

Article • 08/03/2021

Computes hash code for a value.

Syntax

C++

```
template <class Ty>
struct hash {
    size_t operator()(Ty val) const;
};
```

Remarks

The function object defines a hash function, suitable for mapping values of type *Ty* to a distribution of index values. The member `operator()` returns a hash code for *val*, suitable for use with class templates `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. The standard library provides specializations for basic types: *Ty* may be any scalar type, including pointer types and enumeration types. In addition, there are specializations for the library types `string`, `wstring`, `u16string`, `u32string`, `string_view`, `wstring_view`, `u16string_view`, `u32string_view`, `bitset`, `error_code`, `error_condition`, `optional`, `shared_ptr`, `thread`, `type_index`, `unique_ptr`, `variant`, and `vector<bool>`.

Example

C++

```
// std__functional__hash.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>
#include <unordered_set>

int main()
    {
    std::unordered_set<int, std::hash<int> > c0;
    c0.insert(3);
    std::cout << *c0.find(3) << std::endl;
```

```
    return (0);
    }
```

Output

```
3
```

Requirements

Header: <functional>

Namespace: std

See also

<unordered_map>
unordered_multimap Class
unordered_multiset Class
<unordered_set>

is_bind_expression Class

Article • 08/03/2021

Tests if type generated by calling `bind`.

Syntax

C++

```
template<class Ty>
struct is_bind_expression {
   static const bool value;
};
```

Remarks

The constant member `value` is true if the type `Ty` is a type returned by a call to `bind`, otherwise false.

Example

C++

```
// std__functional__is_bind_expression.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

void square(double x)
{
    std::cout << x << "^2 == " << x * x << std::endl;
}

template<class Expr>
void test_for_bind(const Expr&)
{
    std::cout << std::is_bind_expression<Expr>::value << std::endl;
}

int main()
{
    test_for_bind(3.0 * 3.0);
    test_for_bind(std::bind(square, 3));
```

```
    return (0);
}
```

Output

```
0
1
```

is_placeholder Class

Article • 10/20/2023

Tests if type is a placeholder.

Syntax

C++

```
struct is_placeholder {
   static const int value;
};
```

Remarks

The constant value `value` is 0 if the type `Ty` is not a placeholder; otherwise, its value is the position of the function call argument that it binds to. You use it to determine the value `N` for the Nth placeholder `_N`.

Example

C++

```
// std__functional__is_placeholder.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

using namespace std::placeholders;

template<class Expr>
void test_for_placeholder(const Expr&)
{
    std::cout << std::is_placeholder<Expr>::value << std::endl;
}

int main()
{
    test_for_placeholder(3.0);
    test_for_placeholder(_3);

    return (0);
}
```

Output

```
0
3
```

less Struct

Article • 08/03/2021

A binary predicate that performs the less-than operation (`operator<`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct less : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator<
template <>
struct less<void>
{
    template <class T, class U>
    auto operator()(T&& Left, U&& Right) const
        -> decltype(std::forward<T>(Left) <std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator<` that takes operands of the specified or inferred types.

Left
The left operand of the less-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the less-than operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left < Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator<`.

Remarks

The binary predicate `less<Type>` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

C++

```
// functional_less.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

struct MyStruct {
   MyStruct(int i) : m_i(i){}

   bool operator < (const MyStruct & rhs) const {
      return m_i < rhs.m_i;
   }

   int m_i;
};

int main() {
   using namespace std;
   vector <MyStruct> v1;
   vector <MyStruct>::iterator Iter1;
   vector <MyStruct>::reverse_iterator rIter1;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
       v1.push_back( MyStruct(rand()));

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
cout << Iter1->m_i << " ";
   cout << ")" << endl;

   // To sort in ascending order,
```

```
   sort( v1.begin( ), v1.end( ), less<MyStruct>());

   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin() ; Iter1 != v1.end() ; Iter1++ )
cout << Iter1->m_i << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = (41 18467 6334 26500 19169 15724 11478)
Sorted vector v1 = (41 6334 11478 15724 18467 19169 26500)
```

less_equal Struct

Article • 08/03/2021

A binary predicate that performs the less-than-or-equal-to operation (`operator<=`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct less_equal : public binary_function <Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator<=
template <>
struct less_equal<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const
    -> decltype(std::forward<T>(Left) <= std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator<=` that takes operands of the specified or inferred types.

Left
The left operand of the less-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the less-than-or-equal-to operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left <= Right`. The specialized template does perfect forwarding of the result, which has the type returned by `operator<=`.

Remarks

The binary predicate `less_equal<Type>` provides a strict weak ordering of a set of element values of type *Type* into equivalence classes, if and only if this type satisfies the standard mathematical requirements for being so ordered. The specializations for any pointer type yield a total ordering of elements, in that all elements of distinct values are ordered with respect to each other.

Example

C++

```
// functional_less_equal.cpp
// compile with: /EHsc
#define _CRT_RAND_S
#include <stdlib.h>
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter1;
   vector <int>::reverse_iterator rIter1;
   unsigned int randomNumber;

   int i;
   for ( i = 0 ; i < 5 ; i++ )
   {
      if ( rand_s( &randomNumber ) == 0 )
      {
         // Convert the random number to be between 1 - 50000
         // This is done for readability purposes
         randomNumber = ( unsigned int) ((double)randomNumber /
            (double) UINT_MAX * 50000) + 1;

         v1.push_back( randomNumber );
      }
   }
   for ( i = 0 ; i < 3 ; i++ )
   {
```

```
      v1.push_back( 2836 );
   }

   cout << "Original vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // To sort in ascending order,
   // use the binary predicate less_equal<int>( )
   sort( v1.begin( ), v1.end( ), less_equal<int>( ) );
   cout << "Sorted vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;
}
```

Output

```
Original vector v1 = (31247 37154 48755 15251 6205 2836 2836 2836)
Sorted vector v1 = (2836 2836 2836 6205 15251 31247 37154 48755)
```

logical_and Struct

Article • 08/03/2021

A predefined function object that performs the logical conjunction operation (`operator&&`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct logical_and : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator&&
template <>
struct logical_and<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) && std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator&&` that takes operands of the specified or inferred types.

Left
The left operand of the logical conjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the logical conjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left && Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator&&`.

Remarks

For user-defined types, there is no short-circuiting of operand evaluation. Both arguments are evaluated by `operator&&`.

Example

C++

```
// functional_logical_and.cpp
// compile with: /EHsc

#define _CRT_RAND_S
#include <stdlib.h>
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque<bool> d1, d2, d3( 7 );
   deque<bool>::iterator iter1, iter2, iter3;

   unsigned int randomValue;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      if ( rand_s( &randomValue ) == 0 )
      {
         d1.push_back((bool)(( randomValue % 2 ) != 0));
      }

   }

   int j;
   for ( j = 0 ; j < 7 ; j++ )
   {
      if ( rand_s( &randomValue ) == 0 )
      {
         d2.push_back((bool)(( randomValue % 2 ) != 0));
      }
   }
```

```
   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   cout << "Original deque:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;

   // To find element-wise conjunction of the truth values
   // of d1 & d2, use the logical_and function object
   transform( d1.begin( ), d1.end( ), d2.begin( ),
      d3.begin( ), logical_and<bool>( ) );
   cout << "The deque which is the conjunction of d1 & d2 is:\n d3 = ( " ;
   for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
      cout << *iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
Original deque:
d1 = ( true true true true true false false )
Original deque:
d2 = ( true false true true false true false )
The deque which is the conjunction of d1 & d2 is:
d3 = ( true false true true false false false )
```

logical_not Struct

Article • 08/03/2021

A predefined function object that performs the logical not operation (`operator!`) on its argument.

Syntax

```
template <class Type = void>
struct logical_not : public unary_function<Type, bool>
{
    bool operator()(const Type& Left) const;
};

// specialized transparent functor for operator!
template <>
struct logical_not<void>
{
  template <class Type>
  auto operator()(Type&& Left) const`
     -> decltype(!std::forward<Type>(Left));
};
```

Parameters

Type
Any type that supports an `operator!` that takes an operand of the specified or inferred type.

Left
The operand of the logical not operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *Type*.

Return Value

The result of `!Left`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator!`.

Example

C++

```
// functional_logical_not.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque<bool> d1, d2 ( 7 );
   deque<bool>::iterator iter1, iter2;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      d1.push_back((bool)((i % 2) != 0));
   }

   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   // To flip all the truth values of the elements,
   // use the logical_not function object
   transform( d1.begin( ), d1.end( ), d2.begin( ),logical_not<bool>( ) );
   cout << "The deque with its values negated is:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;
}
```

Output

```
Original deque:
d1 = ( false true false true false true false )
The deque with its values negated is:
d2 = ( true false true false true false true )
```

logical_or Struct

Article • 08/03/2021

A predefined function object that performs the logical disjunction operation (`operator||`) on its arguments.

Syntax

```
template <class Type = void>
struct logical_or : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator||
template <>
struct logical_or<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) || std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator||` that takes operands of the specified or inferred types.

Left
The left operand of the logical disjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the logical disjunction operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left || Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator||`.

Remarks

For user-defined types, there is no short-circuiting of operand evaluation. Both arguments are evaluated by `operator||`.

Example

C++

```
// functional_logical_or.cpp
// compile with: /EHsc
#include <deque>
#include <algorithm>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;
   deque <bool> d1, d2, d3( 7 );
   deque <bool>::iterator iter1, iter2, iter3;

   int i;
   for ( i = 0 ; i < 7 ; i++ )
   {
      d1.push_back((bool)((rand() % 2) != 0));
   }

   int j;
   for ( j = 0 ; j < 7 ; j++ )
   {
      d2.push_back((bool)((rand() % 2) != 0));
   }

   cout << boolalpha;    // boolalpha I/O flag on

   cout << "Original deque:\n d1 = ( " ;
   for ( iter1 = d1.begin( ) ; iter1 != d1.end( ) ; iter1++ )
      cout << *iter1 << " ";
   cout << ")" << endl;

   cout << "Original deque:\n d2 = ( " ;
   for ( iter2 = d2.begin( ) ; iter2 != d2.end( ) ; iter2++ )
      cout << *iter2 << " ";
   cout << ")" << endl;
```

```
   // To find element-wise disjunction of the truth values
   // of d1 & d2, use the logical_or function object
   transform( d1.begin( ), d1.end( ), d2.begin( ),
      d3.begin( ), logical_or<bool>( ) );
   cout << "The deque which is the disjuction of d1 & d2 is:\n d3 = ( " ;
   for ( iter3 = d3.begin( ) ; iter3 != d3.end( ) ; iter3++ )
      cout << *iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
Original deque:
d1 = ( true true false false true false false )
Original deque:
d2 = ( false false false true true true true )
The deque which is the disjuction of d1 & d2 is:
d3 = ( true true false true true true true )
```

mem_fun_ref_t Class

Article • 08/03/2021

An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type>
class mem_fun_ref_t : public unary_function<Type, Result> {
    explicit mem_fun_ref_t(
    Result (Type::* _Pm)());

    Result operator()(Type& left) const;
};
```

Parameters

_Pm
A pointer to the member function of class `Type` to be converted to a function object.

left
The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The class template stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (**left**.* `_Pm`)().

Example

The constructor of `mem_fun_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

mem_fun_t Class

Article • 08/03/2021

An adapter class that allows a `non_const` member function that takes no arguments to be called as a unary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

```
C++
```

```
template <class Result, class Type>
class mem_fun_t : public unary_function<Type *, Result> {
    explicit mem_fun_t(Result (Type::* _Pm)());

    Result operator()(Type* _Pleft) const;
};
```

Parameters

_Pm
A pointer to the member function of class `Type` to be converted to a function object.

_Pleft
The object that the *_Pm* member function is called on.

Return Value

An adaptable unary function.

Remarks

The class template stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (`_Pleft`->* `_Pm`)().

Example

The constructor of `mem_fun_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

mem_fun1_ref_t Class

Article • 08/03/2021

An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a reference argument. Deprecated in C++11, removed in C++17.

Syntax

```
C++
```

```
template <class Result, class Type, class Arg>
class mem_fun1_ref_t : public binary_function<Type, Arg, Result> {
    explicit mem_fun1_ref_t(
    Result (Type::* _Pm)(Arg));

    Result operator()(
    Type& left,
    Arg right) const;
};
```

Parameters

_Pm
A pointer to the member function of class `Type` to be converted to a function object.

left
The object that the *_Pm* member function is called on.

right
The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The class template stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (**left**.* `_Pm`)(**right**).

Example

The constructor of `mem_fun1_ref_t` is not usually used directly; the helper function `mem_fun_ref` is used to adapt member functions. See mem_fun_ref for an example of how to use member function adaptors.

mem_fun1_t Class

Article • 08/03/2021

An adapter class that allows a `non_const` member function that takes a single argument to be called as a binary function object when initialized with a pointer argument. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Result, class Type, class Arg>
class mem_fun1_t : public binary_function<Type *, Arg, Result> {
    explicit mem_fun1_t(
    Result (Type::* _Pm)(Arg));

    Result operator()(
    Type* _Pleft,
    Arg right) const;
};
```

Parameters

_Pm
A pointer to the member function of class `Type` to be converted to a function object.

_Pleft
The object that the *_Pm* member function is called on.

right
The argument that is being given to *_Pm*.

Return Value

An adaptable binary function.

Remarks

The class template stores a copy of *_Pm*, which must be a pointer to a member function of class `Type`, in a private member object. It defines its member function `operator()` as returning (**_Pleft**->* `_Pm`)(**right**).

Example

The constructor of `mem_fun1_t` is not usually used directly; the helper function `mem_fun` is used to adapt member functions. See mem_fun for an example of how to use member function adaptors.

minus Struct

Article • 08/03/2021

A predefined function object that performs the subtraction operation (binary `operator-`) on its arguments.

Syntax

```
template <class Type = void>
struct minus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator-
template <>
struct minus<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) - std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
A type that supports a binary `operator-` that takes operands of the specified or inferred types.

Left
The left operand of the operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left - Right`. The specialized template does perfect forwarding of the result, which has the type returned by `operator-`.

Example

C++

```
// functional_minus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
   {
      v1.push_back( 4 * i + 1);
   }

   int j;
   for ( j = 0 ; j <= 5 ; j++ )
   {
      v2.push_back( 3 * j - 1);
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise diference of the elements of v1 & v2
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
      minus<int>( ) );

   cout << "The element-wise differences between v1 and v2 are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 1 5 9 13 17 21 )
The vector v2 = ( -1 2 5 8 11 14 )
The element-wise differences between v1 and v2 are: ( 2 3 4 5 6 7 )
```

modulus Struct

Article • 08/03/2021

A predefined function object that performs the modulus division operation (`operator%`) on its arguments.

Syntax

```
template <class Type = void>
struct modulus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator%
template <>
struct modulus<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) % std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator%` that takes operands of the specified or inferred types.

Left
The left operand of the modulus operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the modulus operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left % Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator%`.

Remarks

The `modulus` functor is restricted to integral types for the basic data types, or to user-defined types that implement `operator%`.

Example

C++

```
// functional_modulus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 1 ; i <= 6 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 3 * j );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise remainders of the elements of v1 & v2
```

```
   transform (v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
      modulus<int>() );

   cout << "The element-wise remainders of the modular division\n are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 5 10 15 20 25 30 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise remainders of the modular division
are: ( 2 4 6 8 10 12 )
```

multiplies Struct

Article • 08/03/2021

A predefined function object that performs the multiplication operation (binary `operator*`) on its arguments.

Syntax

```
template <class Type = void>
struct multiplies : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator*
template <>
struct multiplies<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) * std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
A type that supports a binary `operator*` that takes operands of the specified or inferred types.

Left
The left operand of the multiplication operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the multiplication operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left * Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator*`.

Example

C++

```
// functional_multiplies.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2, v3 ( 6 );
   vector <int>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 1 ; i <= 6 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   int j;
   for ( j = 1 ; j <= 6 ; j++ )
   {
      v2.push_back( 3 * j );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise products of the elements of v1 & v2
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), v3.begin ( ),
      multiplies<int>( ) );

   cout << "The element-wise products of vectors V1 & v2\n are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 2 4 6 8 10 12 )
The vector v2 = ( 3 6 9 12 15 18 )
The element-wise products of vectors V1 & v2
are: ( 6 24 54 96 150 216 )
```

negate Struct

Article • 08/03/2021

A predefined function object that performs the arithmetic negation operation (unary `operator-`) on its argument.

Syntax

```
template <class Type = void>
struct negate : public unary_function<Type, Type>
{
    Type operator()(const Type& Left) const;
};

// specialized transparent functor for unary operator-
template <>
struct negate<void>
{
  template <class Type>
  auto operator()(Type&& Left) const`
    -> decltype(-std::forward<Type>(Left));
};
```

Parameters

Type
Any type that supports an `operator-` that takes an operand of the specified or inferred type.

Left
The operand to be negated. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *Type*.

Return Value

The result of `-Left`. The specialized template does perfect forwarding of the result, which has the type that's returned by unary `operator-`.

Example

C++

```cpp
// functional_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <int> v1, v2 ( 8 );
   vector <int>::iterator Iter1, Iter2;

   int i;
   for ( i = -2 ; i <= 5 ; i++ )
   {
      v1.push_back( 5 * i );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   // Finding the element-wise negatives of the vector v1
   transform ( v1.begin( ),  v1.end( ), v2.begin( ), negate<int>( ) );

   cout << "The negated elements of the vector = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( -10 -5 0 5 10 15 20 25 )
The negated elements of the vector = ( 10 5 0 -5 -10 -15 -20 -25 )
```

not_equal_to Struct

Article • 08/03/2021

A binary predicate that performs the inequality operation (`operator!=`) on its arguments.

Syntax

```
template <class Type = void>
struct not_equal_to : public binary_function<Type, Type, bool>
{
    bool operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator!=
template <>
struct not_equal_to<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) != std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
Any type that supports an `operator!=` that takes operands of the specified or inferred types.

Left
The left operand of the inequality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the inequality operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left != Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by `operator!=`.

Remarks

The objects of type *Type* must be equality-comparable. This requires that the `operator!=` defined on the set of objects satisfies the mathematical properties of an equivalence relation. All of the built-in numeric and pointer types satisfy this requirement.

Example

C++

```
// functional_not_equal_to.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 (6);
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i+=2 )
   {
      v1.push_back( 2.0 *i );
      v1.push_back( 2.0 * i + 1.0 );
   }

   int j;
   for ( j = 0 ; j <= 5 ; j+=2 )
   {
      v2.push_back( - 2.0 * j );
      v2.push_back( 2.0 * j + 1.0 );
   }

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
```

```
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Testing for the element-wise equality between v1 & v2
   transform ( v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
      not_equal_to<double>( ) );

   cout << "The result of the element-wise not_equal_to comparsion\n"
      << "between v1 & v2 is: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 0 1 4 5 8 9 )
The vector v2 = ( -0 1 -4 5 -8 9 )
The result of the element-wise not_equal_to comparsion
between v1 & v2 is: ( 0 0 1 0 1 0 )
```

plus Struct

Article • 08/03/2021

A predefined function object that performs the addition operation (binary `operator+`) on its arguments.

Syntax

C++

```
template <class Type = void>
struct plus : public binary_function <Type, Type, Type>
{
    Type operator()(const Type& Left, const Type& Right) const;
};

// specialized transparent functor for operator+
template <>
struct plus<void>
{
  template <class T, class U>
  auto operator()(T&& Left, U&& Right) const`
    -> decltype(std::forward<T>(Left) + std::forward<U>(Right));
};
```

Parameters

Type, *T*, *U*
A type that supports a binary `operator+` that takes operands of the specified or inferred types.

Left
The left operand of the addition operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *T*.

Right
The right operand of the addition operation. The unspecialized template takes an lvalue reference argument of type *Type*. The specialized template does perfect forwarding of lvalue and rvalue reference arguments of inferred type *U*.

Return Value

The result of `Left + Right`. The specialized template does perfect forwarding of the result, which has the type that's returned by binary `operator+`.

Example

C++

```
// functional_plus.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main( )
{
   vector <double> v1, v2, v3 ( 6 );
   vector <double>::iterator Iter1, Iter2, Iter3;

   int i;
   for ( i = 0 ; i <= 5 ; i++ )
      v1.push_back( 4 * i );

   int j;
   for ( j = 0 ; j <= 5 ; j++ )
      v2.push_back( -2.0 * j - 4 );

   cout << "The vector v1 = ( " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << ")" << endl;

   cout << "The vector v2 = ( " ;
   for ( Iter2 = v2.begin( ) ; Iter2 != v2.end( ) ; Iter2++ )
      cout << *Iter2 << " ";
   cout << ")" << endl;

   // Finding the element-wise sums of the elements of v1 & v2
   transform (v1.begin( ), v1.end( ), v2.begin( ), v3.begin ( ),
plus<double>( ) );

   cout << "The element-wise sums are: ( " ;
   for ( Iter3 = v3.begin( ) ; Iter3 != v3.end( ) ; Iter3++ )
      cout << *Iter3 << " ";
   cout << ")" << endl;
}
```

Output

```
The vector v1 = ( 0 4 8 12 16 20 )
The vector v2 = ( -4 -6 -8 -10 -12 -14 )
The element-wise sums are: ( -4 -2 0 2 4 6 )
```

pointer_to_binary_function Class

Article • 08/03/2021

Converts a binary function pointer into an adaptable binary function. Deprecated in C++11, removed in C++17.

Syntax

C++

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function
    : public binary_function <Arg1, Arg2, Result>
{
    explicit pointer_to_binary_function(
        Result(*pfunc)(Arg1, Arg2));
    Result operator()(Arg1 left, Arg2 right) const;
};
```

Parameters

pfunc
The binary function to be converted.

left
The left object that the **pfunc* is called on.

right
The right object that the **pfunc* is called on.

Return Value

The class template stores a copy of `pfunc`. It defines its member function `operator()` as returning `(* pfunc)(Left, right)`.

Remarks

A binary function pointer is a function object and may be passed to any C++ Standard Library algorithm that is expecting a binary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a

negator, it must be supplied with the nested types `first_argument_type`, `second_argument_type`, and `result_type` that make such an adaptation possible. The conversion by `pointer_to_binary_function` allows the function adaptors to work with binary function pointers.

Example

The constructor of `pointer_to_binary_function` is rarely used directly. See the helper function ptr_fun for an example of how to declare and use the `pointer_to_binary_function` adaptor predicate.

pointer_to_unary_function Class

Article • 08/03/2021

Converts a unary function pointer into an adaptable unary function. Deprecated in C++11, removed in C++17.

Syntax

```
C++
```

```
template <class Arg, class Result>
class pointer_to_unary_function
    : public unary_function<Arg, Result>
{
    explicit pointer_to_unary_function(Result(*pfunc)(Arg));
    Result operator()(Arg left) const;
};
```

Parameters

pfunc
The binary function to be converted.

left
The object that the **pfunc* is called on.

Return Value

The class template stores a copy of `pfunc`. It defines its member function `operator()` as returning (* **pfunc**)(_ *Left*).

Remarks

A unary function pointer is a function object and may be passed to any C++ Standard Library algorithm that is expecting a unary function as a parameter, but it is not adaptable. To use it with an adaptor, such as binding a value to it or using it with a negator, it must be supplied with the nested types `argument_type` and `result_type` that make such an adaptation possible. The conversion by `pointer_to_unary_function` allows the function adaptors to work with binary function pointers.

Example

The constructor of `pointer_to_unary_function` is rarely used directly. See the helper function ptr_fun for an example of how to declare and use the `pointer_to_unary_function` adaptor predicate.

reference_wrapper Class

Article • 11/22/2021

Wraps a reference.

Syntax

C++

```
template <class Ty>
class reference_wrapper
{
    typedef Ty type;

    reference_wrapper(Ty&) noexcept;
    operator Ty&() const noexcept;
    Ty& get() const noexcept;

    template <class... Types>
    auto operator()(Types&&... args) const ->
        decltype(std::invoke(get(), std::forward<Types>(args)...));
};
```

Remarks

A `reference_wrapper<Ty>` is a copy constructible and copy assignable wrapper around a reference to an object or a function of type `Ty`, and holds a pointer that points to an object of that type. A `reference_wrapper` can be used to store references in standard containers, and to pass objects by reference to `std::bind`.

The type `Ty` must be an object type or a function type, or a static assert fails at compile time.

The helper functions std::ref and std::cref can be used to create `reference_wrapper` objects.

Members

Constructors

Name	Description

Name	Description
reference_wrapper	Constructs a `reference_wrapper`.

Typedefs

Name	Description
result_type	The weak result type of the wrapped reference.
type	The type of the wrapped reference.

Functions

Name	Description
get	Obtains the wrapped reference.

Operators

Name	Description
operator Ty&	Gets a pointer to the wrapped reference.
operator()	Calls the wrapped reference.

get

Obtains the wrapped reference.

```
C++
```

```
Ty& get() const noexcept;
```

Remarks

The member function returns the wrapped reference.

Example

```
C++
```

```cpp
// std__functional__reference_wrapper_get.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int main() {
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << rwi << std::endl;
    rwi.get() = -1;
    std::cout << "i = " << i << std::endl;

    return (0);
}
```

Output

```
i = 1
rwi = 1
i = -1
```

operator Ty&

Gets the wrapped reference.

C++

```cpp
operator Ty&() const noexcept;
```

Remarks

The member operator returns `*ptr`.

Example

C++

```cpp
// std__functional__reference_wrapper_operator_cast.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int main() {
```

```
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "(int)rwi = " << (int)rwi << std::endl;

    return (0);
}
```

Output

```
i = 1
(int)rwi = 1
```

operator()

Calls the wrapped reference.

C++

```
template <class... Types>
auto operator()(Types&&... args);
```

Parameters

Types
The argument list types.

args
The argument list.

Remarks

The template member `operator()` returns `std::invoke(get(), std::forward<Types>(args)...)`.

Example

C++

```
// std__functional__reference_wrapper_operator_call.cpp
// compile with: /EHsc
#include <functional>
```

```cpp
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    std::reference_wrapper<int (int)> rwi(neg);

    std::cout << "rwi(3) = " << rwi(3) << std::endl;

    return (0);
}
```

Output

```
rwi(3) = -3
```

reference_wrapper

Constructs a `reference_wrapper`.

C++

```cpp
reference_wrapper(Ty& val) noexcept;
```

Parameters

Ty
The type to wrap.

val
The value to wrap.

Remarks

The constructor sets the stored value `ptr` to `&val`.

Example

C++

```cpp
// std__functional__reference_wrapper_reference_wrapper.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    int i = 1;
    std::reference_wrapper<int> rwi(i);

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << rwi << std::endl;
    rwi.get() = -1;
    std::cout << "i = " << i << std::endl;

    return (0);
}
```

Output

```
i = 1
rwi = 1
i = -1
```

result_type

The weak result type of the wrapped reference.

C++

```cpp
typedef R result_type;
```

Remarks

The `result_type` typedef is a synonym for the weak result type of a wrapped function. This typedef is only meaningful for function types.

Example

C++

```cpp
// std__functional__reference_wrapper_result_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
    return (-val);
}

int main() {
    typedef std::reference_wrapper<int (int)> Mywrapper;
    Mywrapper rwi(neg);
    Mywrapper::result_type val = rwi(3);

    std::cout << "val = " << val << std::endl;

    return (0);
}
```

Output

```
val = -3
```

type

The type of the wrapped reference.

C++

```cpp
typedef Ty type;
```

Remarks

The typedef is a synonym for the template argument `Ty`.

Example

C++

```cpp
// std__functional__reference_wrapper_type.cpp
// compile with: /EHsc
#include <functional>
#include <iostream>

int neg(int val) {
```

```
    return (-val);
}

int main() {
    int i = 1;
    typedef std::reference_wrapper<int> Mywrapper;
    Mywrapper rwi(i);
    Mywrapper::type val = rwi.get();

    std::cout << "i = " << i << std::endl;
    std::cout << "rwi = " << val << std::endl;

    return (0);
}
```

Output

```
i = 1
rwi = 1
```

unary_function Struct

Article • 08/03/2021

An empty base struct that defines types that may be inherited by derived classes that provides a unary function object.

Syntax

C++

```
struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type;
};
```

Remarks

The template struct serves as a base for classes that define a member function of the form `result_type operator()( constargument_type& ) const`.

All such derived unary functions can refer to their sole argument type as **argument_type** and their return type as **result_type**.

Example

C++

```
// functional_unary_function.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

// Creation of a user-defined function object
// that inherits from the unary_function base class
class greaterthan10: unary_function<int, bool>
{
public:
    result_type operator()(argument_type i)
    {
```

```
        return (result_type)(i > 10);
    }
};

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 5; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( " ;
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), greaterthan10());
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 )
The number of elements in v1 greater than 10 is: 3.
```

unary_negate Class

Article • 08/03/2021

A class template providing a member function that negates the return value of a specified unary function. Deprecated in C++17 in favor of not_fn.

Syntax

```
C++
```

```
template <class Predicate>
class unary_negate
    : public unaryFunction<typename Predicate::argument_type, bool>
{
    explicit unary_negate(const Predicate& Func);
    bool operator()(const typename Predicate::argument_type& left) const;
};
```

Parameters

Func
The unary function to be negated.

left
The operand of the unary function to be negated.

Return Value

The negation of the unary function.

Remarks

The class template stores a copy of a unary function object *_Func*. It defines its member function `operator()` as returning `!_Func(left)`.

The constructor of `unary_negate` is rarely used directly. The helper function not1 provides an easier way to declare and use the **unary_negator** adaptor predicate.

Example

C++

```
// functional_unary_negate.cpp
// compile with: /EHsc
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v1;
    vector<int>::iterator Iter;

    int i;
    for (i = 0; i <= 7; i++)
    {
        v1.push_back(5 * i);
    }

    cout << "The vector v1 = ( ";
    for (Iter = v1.begin(); Iter != v1.end(); Iter++)
        cout << *Iter << " ";
    cout << ")" << endl;

    vector<int>::iterator::difference_type result1;
    // Count the elements greater than 10
    result1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;

    vector<int>::iterator::difference_type result2;
    // Use the negator to count the elements less than or equal to 10
    result2 = count_if(v1.begin(), v1.end(),
        unary_negate<binder2nd <greater<int> > >(bind2nd(greater<int>
(),10)));

    // The following helper function not1 also works for the above line
    // not1(bind2nd(greater<int>(), 10)));

    cout << "The number of elements in v1 not greater than 10 is: "
         << result2 << "." << endl;
}
```

Output

```
The vector v1 = ( 0 5 10 15 20 25 30 35 )
The number of elements in v1 greater than 10 is: 5.
The number of elements in v1 not greater than 10 is: 3.
```

<future>

Article • 10/17/2022

Include the standard header <future> to define class templates and supporting templates that simplify running a function—possibly in a separate thread—and retrieving its result. The result is either the value that is returned by the function or an exception that is emitted by the function but is not caught in the function.

This header uses Concurrency Runtime (ConcRT) so that you can use it together with other ConcRT mechanisms. For more information about ConcRT, see Concurrency Runtime.

Syntax

```
C++

#include <future>
```

Remarks

> **Note**
>
> In code that is compiled by using **/clr**, this header is blocked.

An *asynchronous provider* stores the result of a function call. An *asynchronous return object* is used to retrieve the result of a function call. An *associated asynchronous state* provides communication between an asynchronous provider and one or more asynchronous return objects.

A program does not directly create any associated asynchronous state objects. The program creates an asynchronous provider whenever it needs one and from that it creates an asynchronous return object that shares its associated asynchronous state with the provider. Asynchronous providers and asynchronous return objects manage the objects that hold their shared associated asynchronous state. When the last object that references the associated asynchronous state releases it, the object that holds the associated asynchronous state is destroyed.

An asynchronous provider or an asynchronous return object that has no associated asynchronous state is *empty*.

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

The template function `async` and the class templates `promise` and `packaged_task` are asynchronous providers. The class templates `future` and `shared_future` describe asynchronous return objects.

Each of the class templates `promise`, `future`, and `shared_future` has a specialization for the type **`void`** and a partial specialization for storing and retrieving a value by reference. These specializations differ from the primary template only in the signatures and semantics of the functions that store and retrieve the returned value.

The class templates `future` and `shared_future` never block in their destructors, except in one case that's preserved for backward compatibility: Unlike all other futures, for a `future`—or the last `shared_future`—that's attached to a task started with `std::async`, the destructor blocks if the task has not completed; that is, it blocks if this thread did not yet call `.get()` or `.wait()` and the task is still running. The following usability note has been added to the description of `std::async` in the draft standard: "[Note: If a future obtained from std::async is moved outside the local scope, other code that uses the future must be aware that the future's destructor may block for the shared state to become ready.—end note]" In all other cases, `future` and `shared_future` destructors are required and are guaranteed to never block.

Members

Classes

Name	Description
future Class	Describes an asynchronous return object.
future_error Class	Describes an exception object that can be thrown by methods of types that manage `future` objects.
packaged_task Class	Describes an asynchronous provider that is a call wrapper and whose call signature is `Ty(ArgTypes...)`. Its associated asynchronous state holds a copy of its callable object in addition to the potential result.
promise Class	Describes an asynchronous provider.
shared_future Class	Describes an asynchronous return object. In contrast with a `future` object, an asynchronous provider can be associated with any number of `shared_future` objects.

Structures

Name	Description
is_error_code_enum Structure	Specialization that indicates that `future_errc` is suitable for storing an `error_code`.
uses_allocator Structure	Specialization that always holds true.

Functions

Name	Description
async	Represents an asynchronous provider.
future_category	Returns a reference to the `error_category` object that characterizes errors that are associated with `future` objects.
make_error_code	Creates an `error_code` that has the `error_category` object that characterizes `future` errors.
make_error_condition	Creates an `error_condition` that has the `error_category` object that characterizes `future` errors.
swap	Exchanges the associated asynchronous state of one `promise` object with that of another.

Enumerations

Name	Description
future_errc	Supplies symbolic names for the errors that are reported by the `future_error` class.
future_status	Supplies symbolic names for the reasons that a timed wait function can return.
launch	Represents a bitmask type that describes the possible modes for the template function `async`.

See also

Header Files Reference

`<future>` functions

Article • 09/13/2024

async
future_category
make_error_code
make_error_condition
swap

async

Represents an *asynchronous provider*.

C++

```
template <class Fn, class... ArgTypes>
future<typename result_of<Fn(ArgTypes...)>::type>
    async(Fn&& fn, ArgTypes&&... args);

template <class Fn, class... ArgTypes>
future<typename result_of<Fn(ArgTypes...)>::type>
    async(launch policy, Fn&& fn, ArgTypes&&... args);
```

Parameters

`policy`
A launch value.

Remarks

Definitions of abbreviations:

Expand table

Abbreviation	Description
`dfn`	The result of calling `decay_copy(forward<Fn>(fn))`.
`dargs`	The results of the calls `decay_copy(forward<ArgsTypes>(args...))`.
`Ty`	The type `result_of<Fn(ArgTypes...)>::type`.

The first template function returns `async(launch::any, fn, args...)`.

The second function returns a `future<Ty>` object whose *associated asynchronous state* holds a result together with the values of *`dfn`* and *`dargs`* and a thread object to manage a separate thread of execution.

Unless `decay<Fn>::type` is a type other than launch, the second function doesn't participate in overload resolution.

The C++ standard states that if the policy is `launch::async`, the function behaves as if it invokes the callable object in a new thread. This means that while it typically results in creating a new thread, the implementation may use other mechanisms to achieve equivalent behavior. However, the Microsoft implementation currently doesn't conform strictly to this behavior. It obtains threads from the Windows ThreadPool, which may provide a recycled thread rather than a new one. This means that the `launch::async` policy is effectively implemented as `launch::async|launch::deferred`. Another implication of the ThreadPool-based implementation is that there's no guarantee that thread-local variables are destroyed when the thread completes. If the thread is recycled and provided to a new call to `async`, the old variables still exist. We recommend that you avoid using thread-local variables with `async`.

If *`policy`* is `launch::deferred`, the function marks its associated asynchronous state as holding a *deferred function* and returns. The first call to any nontimed function that waits for the associated asynchronous state to be ready in effect calls the deferred function by evaluating `INVOKE(dfn, dargs..., Ty)`.

In all cases, the associated asynchronous state of the `future` object isn't set to *ready* until the evaluation of `INVOKE(dfn, dargs..., Ty)` completes, either by throwing an exception or by returning normally. The result of the associated asynchronous state is an exception if one was thrown, or the value the evaluation returns.

> **Note**
>
> For a `future`—or the last shared_future—that's attached to a task started with `std::async`, the destructor blocks if the task has not completed; that is, it blocks if this thread did not yet call `.get()` or `.wait()` and the task is still running. If a `future` obtained from `std::async` is moved outside the local scope, other code that uses it must be aware that its destructor may block for the shared state to become ready.

The pseudo-function `INVOKE` is defined in <functional>.

Microsoft specific

When the passed function is executed asynchronously, it executes on the Windows Thread Pool. For more information, see Thread Pools. The number of concurrent threads is limited to the thread pool default, which is 500 threads.

Before Windows 11 and Windows Server 2022, applications were limited by default to a single processor group having at most 64 logical processors. This limited the number of concurrently executing threads to 64. For more information, see Processor Groups.

Starting with Windows 11 and Windows Server 2022, processes and their threads have processor affinities that by default span all processors in the system and across multiple groups on machines with more than 64 processors. The limit on the number of concurrent threads is now the total number of logical processors in the system.

`future_category`

Returns a reference to the error_category object that characterizes errors that are associated with `future` objects.

```
C++
```

```
const error_category& future_category() noexcept;
```

`make_error_code`

Creates an error_code together with the error_category object that characterizes future errors.

```
C++
```

```
inline error_code make_error_code(future_errc Errno) noexcept;
```

Parameters

`Errno`
A future_errc value that identifies the reported error.

Return Value

`error_code(static_cast<int>(Errno), future_category());`

`make_error_condition`

Creates an error_condition together with the error_category object that characterizes future errors.

```
C++
```

```
inline error_condition make_error_condition(future_errc Errno) noexcept;
```

Parameters

`Errno`

A future_errc value that identifies the reported error.

Return Value

`error_condition(static_cast<int>(Errno), future_category());`

`swap`

Exchanges the *associated asynchronous state* of one `promise` object with that of another.

```
C++
```

```
template <class Ty>
void swap(promise<Ty>& Left, promise<Ty>& Right) noexcept;

template <class Ty, class... ArgTypes>
void swap(packaged_task<Ty(ArgTypes...)>& Left,
packaged_task<Ty(ArgTypes...)>& Right) noexcept;
```

Parameters

`Left`

The left `promise` object.

`Right`

The right `promise` object.

See also

<future>

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`<future>` enums

Article • 12/07/2021

future_errc
future_status
launch

future_errc Enumeration

Supplies symbolic names for all of the errors that are reported by the future_error class.

C++

```
class future_errc {
   broken_promise,
   future_already_retrieved,
   promise_already_satisfied,
   no_state
   };
```

future_status Enumeration

Supplies symbolic names for the reasons that a timed wait function can return.

C++

```
enum future_status{
    ready,
    timeout,
    deferred
};
```

launch Enumeration

Represents a bitmask type that describes the possible modes for the template function async.

C++

```
class launch{
   async,
```

```
    deferred
    };
```

See also

<future>

`future` Class

Article • 06/17/2022

Describes an *asynchronous return object*.

Syntax

```
C++
```

```
template <class Ty>
class future;
```

Remarks

Each standard *asynchronous provider* returns an object whose type is an instantiation of this template. A `future` object provides the only access to the asynchronous provider that it's associated with. If you need multiple asynchronous return objects that are associated with the same asynchronous provider, copy the `future` object to a shared_future object.

Members

Public Constructors

Name	Description
future	Constructs a `future` object.

Public Methods

Name	Description
get	Retrieves the result that is stored in the associated asynchronous state.
share	Converts the object to a `shared_future`.
valid	Specifies whether the object isn't empty.
wait	Blocks the current thread until the associated asynchronous state is ready.

Name	Description
wait_for	Blocks until the associated asynchronous state is ready or until the specified time has elapsed.
wait_until	Blocks until the associated asynchronous state is ready or until a specified point in time.

Public Operators

Name	Description
future::operator=	Transfers the associated asynchronous state from a specified object.

Requirements

Header: `<future>`

Namespace: `std`

Constructors

Constructs a `future` object.

```
C++
```

```
future() noexcept;
future(future&& Other) noexcept;
```

Parameters

`Other`
A `future` object.

Remarks

The first constructor constructs a `future` object that has no associated asynchronous state.

The second constructor constructs a `future` object and transfers the associated asynchronous state from *Other*. *Other* no longer has an associated asynchronous state.

`get`

Retrieves the result that is stored in the associated asynchronous state.

```
C++
```

```
Ty get();
```

Return Value

If the result is an exception, the method rethrows it. Otherwise, the result is returned.

Remarks

Before it retrieves the result, this method blocks the current thread until the associated asynchronous state is ready.

For the partial specialization `future<Ty&>`, the stored value is effectively a reference to the object that was passed to the asynchronous provider as the return value.

Because no stored value exists for the specialization `future<void>`, the method returns `void`.

In other specializations, the method moves its return value from the stored value. Therefore, call this method only once.

`operator=`

Transfers an associated asynchronous state from a specified object.

```
C++
```

```
future& operator=(future&& Right) noexcept;
```

Parameters

Right
A `future` object.

Return Value

`*this`

Remarks

After the transfer, *Right* no longer has an associated asynchronous state.

`share`

Converts the object to a shared_future object.

```
C++
```

```
shared_future<Ty> share();
```

Return Value

`shared_future(move(*this))`

`valid`

Specifies whether the object has an associated asynchronous state.

```
C++
```

```
bool valid() noexcept;
```

Return Value

`true` if the object has an associated asynchronous state; otherwise, **`false`**.

`wait`

Blocks the current thread until the associated asynchronous state is *ready*.

```
C++
```

```
void wait() const;
```

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

`wait_for`

Blocks the current thread until the associated asynchronous state is *ready* or until a specified time interval has elapsed.

```
C++
```

```
template <class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& Rel_time) const;
```

Parameters

`Rel_time`
A chrono::duration object that specifies a maximum time interval that the thread blocks.

Return Value

A future_status that indicates the reason for returning.

Remarks

An associated asynchronous state is ready only if its asynchronous provider has stored a return value or stored an exception.

`wait_until`

Blocks the current thread until the associated asynchronous state is *ready* or until after a specified time point.

```
C++
```

```
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>&
Abs_time) const;
```

Parameters

Abs_time

A time_point object that specifies a time after which the thread can unblock.

Return Value

A future_status that indicates the reason for returning.

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

See also

Header Files Reference
<future>

`future_error` class

Article • 10/11/2021

Describes an exception object that can be thrown by methods of types that manage future objects.

Syntax

```
C++
```

```
class future_error : public logic_error {
public:
    future_error(error_code code);

const error_code& code() const throw();

const char *what() const throw();
};
```

Requirements

Header: `<future>`

Namespace: `std`

See also

Header Files Reference
logic_error class
error_code class

is_error_code_enum Structure

Article • 08/03/2021

Specialization that indicates that future_errc is suitable for storing an error_code.

Syntax

```
C++
```

```
template <>
struct is_error_code_enum<Future_errc> : public true_type;
```

Requirements

Header: <future>

Namespace: std

See also

Header Files Reference

<future>

packaged_task Class

Article • 06/20/2022

Describes an *asynchronous provider* that is a call wrapper whose call signature is `Ty(ArgTypes...)`. Its *associated asynchronous state* holds a copy of its callable object in addition to the potential result.

Syntax

```
C++
```

```
template <class>
class packaged_task;
```

Members

Public Constructors

Name	Description
packaged_task	Constructs a `packaged_task` object.
packaged_task::~packaged_task Destructor	Destroys a `packaged_task` object.

Public Methods

Name	Description
get_future	Returns a future object that has the same associated asynchronous state.
make_ready_at_thread_exit	Calls the callable object that's stored in the associated asynchronous state and atomically stores the returned value.
reset	Replaces the associated asynchronous state.
swap	Exchanges the associated asynchronous state with a specified object.
valid	Specifies whether the object has an associated asynchronous state.

Public Operators

Name	Description
packaged_task::operator=	Transfers an associated asynchronous state from a specified object.
packaged_task::operator()	Calls the callable object that's stored in the associated asynchronous state, atomically stores the returned value, and sets the state to *ready*.
packaged_task::operator bool	Specifies whether the object has an associated asynchronous state.

Requirements

Header: <future>

Namespace: std

packaged_task::get_future

Returns an object of type `future<Ty>` that has the same *associated asynchronous state*.

```
C++

future<Ty> get_future();
```

Remarks

If the `packaged_task` object doesn't have an associated asynchronous state, this method throws a future_error that has an error code of `no_state`.

If this method has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `future_already_retrieved`.

packaged_task::make_ready_at_thread_exit

Calls the callable object that's stored in the *associated asynchronous state* and atomically stores the returned value.

```
C++
```

```
void make_ready_at_thread_exit(ArgTypes... args);
```

Remarks

If the `packaged_task` object doesn't have an associated asynchronous state, this method throws a future_error that has an error code of `no_state`.

If this method or make_ready_at_thread_exit has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `promise_already_satisfied`.

Otherwise, this operator calls `INVOKE(fn, args..., Ty)`, where *fn* is the callable object that's stored in the associated asynchronous state. Any returned value is stored atomically as the returned result of the associated asynchronous state.

In contrast to packaged_task::operator(), the associated asynchronous state isn't set to `ready` until after all thread-local objects in the calling thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state aren't unblocked until the calling thread exits.

packaged_task::operator=

Transfers the *associated asynchronous state* from a specified object.

C++

```
packaged_task& operator=(packaged_task&& Right);
```

Parameters

Right
A `packaged_task` object.

Return Value

`*this`

Remarks

After the operation, *Right* no longer has an associated asynchronous state.

packaged_task::operator()

Calls the callable object that's stored in the *associated asynchronous state*, atomically stores the returned value, and sets the state to *ready*.

```
C++
```

```
void operator()(ArgTypes... args);
```

Remarks

If the `packaged_task` object doesn't have an associated asynchronous state, this method throws a future_error that has an error code of `no_state`.

If this method or make_ready_at_thread_exit has already been called for a `packaged_task` object that has the same associated asynchronous state, the method throws a `future_error` that has an error code of `promise_already_satisfied`.

Otherwise, this operator calls `INVOKE(fn, args..., Ty)`, where *fn* is the callable object that's stored in the associated asynchronous state. Any returned value is stored atomically as the returned result of the associated asynchronous state, and the state is set to ready. As a result, any threads that are blocked on the associated asynchronous state become unblocked.

packaged_task::operator bool

Specifies whether the object has an `associated asynchronous state`.

```
C++
```

```
operator bool() const noexcept;
```

Return Value

`true` if the object has an associated asynchronous state; otherwise, **`false`**.

packaged_task::packaged_task Constructor

Constructs a `packaged_task` object.

```
C++
```

```
packaged_task() noexcept;
packaged_task(packaged_task&& Right) noexcept;
template <class Fn>
   explicit packaged_task(Fn&& fn);

template <class Fn, class Alloc>
   explicit packaged_task(
      allocator_arg_t, const Alloc& alloc, Fn&& fn);
```

Parameters

Right
A `packaged_task` object.

alloc
A memory allocator. For more information, see <allocators>.

fn
A function object.

Remarks

The first constructor constructs a `packaged_task` object that has no *associated asynchronous state*.

The second constructor constructs a `packaged_task` object and transfers the associated asynchronous state from *Right*. After the operation, *Right* no longer has an associated asynchronous state.

The third constructor constructs a `packaged_task` object that has a copy of *fn* stored in its associated asynchronous state.

The fourth constructor constructs a `packaged_task` object that has a copy of *fn* stored in its associated asynchronous state, and uses `alloc` for memory allocation.

packaged_task::~packaged_task Destructor

Destroys a `packaged_task` object.

```
C++

~packaged_task();
```

Remarks

If the *associated asynchronous state* isn't *ready*, the destructor stores a future_error exception that has an error code of `broken_promise` as the result in the associated asynchronous state, and any threads that are blocked on the associated asynchronous state become unblocked.

packaged_task::reset

Uses a new *associated asynchronous state* to replace the existing associated asynchronous state.

```
C++

void reset();
```

Remarks

In effect, this method executes `*this = packaged_task(move(fn))`, where *fn* is the function object that's stored in the associated asynchronous state for this object. Therefore, the state of the object is cleared, and get_future, operator(), and make_ready_at_thread_exit can be called as if on a newly constructed object.

packaged_task::swap

Exchanges the associated asynchronous state with a specified object.

```
C++

void swap(packaged_task& Right) noexcept;
```

Parameters

Right
A `packaged_task` object.

packaged_task::valid

Specifies whether the object has an `associated asynchronous state`.

```
C++
```

```
bool valid() const;
```

Return Value

`true` if the object has an associated asynchronous state; otherwise, **`false`**.

See also

Header Files Reference

<future>

promise Class

Article • 06/20/2022

Describes an *asynchronous provider*.

Syntax

```
C++
```

```
template <class Ty>
class promise;
```

Members

Public Constructors

Name	Description
promise	Constructs a `promise` object.

Public Methods

Name	Description
get_future	Returns a future associated with this promise.
set_exception	Atomically sets the result of this promise to indicate an exception.
set_exception_at_thread_exit	Atomically sets the result of this promise to indicate an exception, and delivers the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).
set_value	Atomically sets the result of this promise to indicate a value.
set_value_at_thread_exit	Atomically sets the result of this promise to indicate a value, and delivers the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).
swap	Exchanges the *associated asynchronous state* of this promise with that of a specified promise object.

Public Operators

Name	Description
promise::operator=	Assignment of the shared state of this promise object.

Inheritance Hierarchy

promise

Requirements

Header: <future>

Namespace: std

promise::get_future

Returns a future object that has the same *associated asynchronous state* as this promise.

```
C++

future<Ty> get_future();
```

Remarks

If the promise object is empty, this method throws a future_error that has an error_code of `no_state`.

If this method has already been called for a promise object that has the same associated asynchronous state, the method throws a `future_error` that has an `error_code` of `future_already_retrieved`.

promise::operator=

Transfers the *associated asynchronous state* from a specified `promise` object.

```
C++

promise& operator=(promise&& Other) noexcept;
```

Parameters

Other
A `promise` object.

Return Value

`*this`

Remarks

This operator transfers the associated asynchronous state from *Other*. After the transfer, *Other* is *empty*.

promise::promise Constructor

Constructs a `promise` object.

```
C++
```

```
promise();
template <class Alloc>
promise(allocator_arg_t, const Alloc& Al);
promise(promise&& Other) noexcept;
```

Parameters

Al
A memory allocator. For more information about allocators, see <allocators>.

Other
A `promise` object.

Remarks

The first constructor constructs an *empty* `promise` object.

The second constructor constructs an empty `promise` object and uses *Al* for memory allocation.

The third constructor constructs a `promise` object and transfers the associated asynchronous state from *Other*, and leaves *Other* empty.

promise::set_exception

Atomically stores an exception as the result of this `promise` object and sets the *associated asynchronous state* to *ready*.

```
C++
```

```
void set_exception(exception_ptr Exc);
```

Parameters

Exc
An exception_ptr that's stored by this method as the exception result.

Remarks

If the `promise` object has no associated asynchronous state, this method throws a future_error that has an error code of `no_state`.

If `set_exception`, set_exception_at_thread_exit, set_value, or set_value_at_thread_exit has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

As a result of this method, any threads that are blocked on the associated asynchronous state become unblocked.

promise::set_exception_at_thread_exit

Atomically sets the result of this `promise` to indicate an exception, delivering the notification only after all thread-local objects in the current thread have been destroyed (usually at thread exit).

```
C++
```

```
void set_exception_at_thread_exit(exception_ptr Exc);
```

Parameters

Exc
An exception_ptr that's stored by this method as the exception result.

Remarks

If the promise object has no *associated asynchronous state*, this method throws a future_error that has an error code of `no_state`.

If set_exception, `set_exception_at_thread_exit`, set_value, or set_value_at_thread_exit has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

In contrast to set_exception, this method doesn't set the associated asynchronous state to ready until after all thread-local objects in the current thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state aren't unblocked until the current thread exits.

promise::set_value

Atomically stores a value as the result of this `promise` object and sets the *associated asynchronous state* to *ready*.

C++

```
void promise::set_value(const Ty& Val);
void promise::set_value(Ty&& Val);
void promise<Ty&>::set_value(Ty& Val);
void promise<void>::set_value();
```

Parameters

Val
The value to be stored as the result.

Remarks

If the `promise` object has no associated asynchronous state, this method throws a future_error that has an error code of `no_state`.

If set_exception, set_exception_at_thread_exit, `set_value`, or set_value_at_thread_exit has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

As a result of this method, any threads that are blocked on the associated asynchronous state become unblocked.

The first method also throws any exception that is thrown when *Val* is copied into the associated asynchronous state. In this situation, the associated asynchronous state isn't set to ready.

The second method also throws any exception that is thrown when *Val* is moved into the associated asynchronous state. In this situation, the associated asynchronous state isn't set to ready.

For the partial specialization `promise<Ty&>`, the stored value is in effect a reference to *Val*.

For the specialization `promise<void>`, no stored value exists.

promise::set_value_at_thread_exit

Atomically stores a value as the result of this `promise` object.

```
C++
```

```
void promise::set_value_at_thread_exit(const Ty& Val);
void promise::set_value_at_thread_exit(Ty&& Val);
void promise<Ty&>::set_value_at_thread_exit(Ty& Val);
void promise<void>::set_value_at_thread_exit();
```

Parameters

Val
The value to be stored as the result.

Remarks

If the promise object has no *associated asynchronous state*, this method throws a future_error that has an error code of `no_state`.

If set_exception, set_exception_at_thread_exit, set_value, or `set_value_at_thread_exit` has already been called for a `promise` object that has the same associated asynchronous state, this method throws a `future_error` that has an error code of `promise_already_satisfied`.

In contrast to `set_value`, the associated asynchronous state isn't set to ready until after all thread-local objects in the current thread have been destroyed. Typically, threads that are blocked on the associated asynchronous state aren't unblocked until the current thread exits.

The first method also throws any exception that is thrown when *Val* is copied into the associated asynchronous state.

The second method also throws any exception that is thrown when *Val* is moved into the associated asynchronous state.

For the partial specialization `promise<Ty&>`, the stored value is effectively a reference to *Val*.

For the specialization `promise<void>`, no stored value exists.

promise::swap

Exchanges the *associated asynchronous state* of this promise object with that of a specified object.

```
C++
```

```
void swap(promise& Other) noexcept;
```

Parameters

Other
A `promise` object.

See also

Header Files Reference

shared_future Class

Article • 06/21/2022

Describes an *asynchronous return object*. In contrast with a future object, an *asynchronous provider* can be associated with any number of `shared_future` objects.

Syntax

```
C++
```

```
template <class Ty>
class shared_future;
```

Remarks

Don't call any methods other than `valid`, `operator=`, and the destructor on a `shared_future` object that's *empty*.

`shared_future` objects aren't synchronized. Calling methods on the same object from multiple threads introduces a data race that has unpredictable results.

Members

Public Constructors

Name	Description
shared_future	Constructs a `shared_future` object.

Public Methods

Name	Description
get	Retrieves the result that's stored in the *associated asynchronous state*.
valid	Specifies whether the object isn't empty.
wait	Blocks the current thread until the associated asynchronous state is ready.

Name	Description
wait_for	Blocks until the associated asynchronous state is ready or until the specified time has elapsed.
wait_until	Blocks until the associated asynchronous state is ready or until a specified point in time.

Public Operators

Name	Description
shared_future::operator=	Assigns a new associated asynchronous state.

Requirements

Header: <future>

Namespace: std

shared_future::get

Retrieves the result that's stored in the *associated asynchronous state*.

```
C++
```

```
const Ty& get() const;

Ty& get() const;

void get() const;
```

Remarks

If the result is an exception, the method rethrows it. Otherwise, the result is returned.

Before it retrieves the result, this method blocks the current thread until the associated asynchronous state is ready.

For the partial specialization `shared_future<Ty&>`, the stored value is effectively a reference to the object that was passed to the *asynchronous provider* as the return value.

Because no stored value exists for the specialization `shared_future<void>`, the method returns `void`.

shared_future::operator=

Transfers an *associated asynchronous state* from a specified object.

```
C++
```

```
shared_future& operator=(shared_future&& Right) noexcept;
shared_future& operator=(const shared_future& Right);
```

Parameters

Right
A `shared_future` object.

Return Value

`*this`

Remarks

For the first operator, *Right* no longer has an associated asynchronous state after the operation.

For the second method, *Right* maintains its associated asynchronous state.

shared_future::shared_future Constructor

Constructs a `shared_future` object.

```
C++
```

```
shared_future() noexcept;
shared_future(future<Ty>&& Right) noexcept;
shared_future(shared_future&& Right) noexcept;
shared_future(const shared_future& Right);
```

Parameters

Right
A future or `shared_future` object.

Remarks

The first constructor constructs a `shared_future` object that has no *associated asynchronous state*.

The second and third constructors construct a `shared_future` object and transfer the associated asynchronous state from *Right*. *Right* no longer has an associated asynchronous state.

The fourth constructor constructs a `shared_future` object that has the same associated asynchronous state as *Right*.

shared_future::valid

Specifies whether the object has an *associated asynchronous state*.

```
C++
```

```
bool valid() noexcept;
```

Return Value

`true` if the object has an associated asynchronous state; otherwise, `false`.

shared_future::wait

Blocks the current thread until the *associated asynchronous state* is *ready*.

```
C++
```

```
void wait() const;
```

Remarks

An associated asynchronous state is ready only if its asynchronous provider has stored a return value or stored an exception.

shared_future::wait_for

Blocks the current thread until the associated asynchronous state is *ready* or until a specified time has elapsed.

```
C++
```

```
template <class Rep, class Period>
future_status wait_for(
    const chrono::duration<Rep, Period>& Rel_time) const;
```

Parameters

Rel_time
A chrono::duration object that specifies a maximum time interval that the thread blocks.

Return Value

A future_status that indicates the reason for returning.

Remarks

An associated asynchronous state is *ready* only if its asynchronous provider has stored a return value or stored an exception.

shared_future::wait_until

Blocks the current thread until the associated asynchronous state is *ready* or until after a specified time point.

```
C++
```

```
template <class Clock, class Duration>
future_status wait_until(
    const chrono::time_point<Clock, Duration>& Abs_time) const;
```

Parameters

Abs_time
A chrono::time_point object that specifies a time after which the thread can unblock.

Return Value

A future_status that indicates the reason for returning.

Remarks

An associated asynchronous state is ready only if its asynchronous provider has stored a return value or stored an exception.

See also

Header Files Reference
<future>

uses_allocator Structure

Article • 08/03/2021

Specializations that always hold true.

Syntax

```
C++
```

```
template <class Ty, class Alloc>
struct uses_allocator<promise<Ty>, Alloc> : true_type;
template <class Ty, class Alloc>
struct uses_allocator<packaged_task<Ty>, Alloc> : true_type;
```

Requirements

Header: <future>

Namespace: std

Specializations

<tuple>

```
C++
```

```
template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;
```

See also

Header Files Reference

<future>

<hash_map>

Article • 12/07/2021

> **Note**
>
> This header is obsolete. The alternative is <unordered_map>.

Defines the container class templates hash_map and hash_multimap and their supporting templates.

Syntax

```
C++

#include <hash_map>
```

Operators

Hash_map version	Hash_multimap version	Description
operator!= (hash_map)	operator!= (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is not equal to the hash_map or hash_multimap object on the right side.
operator== (hash_map)	operator== (hash_multimap)	Tests if the hash_map or hash_multimap object on the left side of the operator is equal to the hash_map or hash_multimap object on the right side.

Specialized Template Functions

Hash_map version	Hash_multimap version	Description
swap (hash_map)	swap (hash_multimap)	Exchanges the elements of two hash_maps or hash_multimaps.

Classes

Class	Description
hash_compare Class	Describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default `Traits` parameter object to order and hash the elements they contain.
value_compare Class	Provides a function object that can compare the elements of a hash_map by comparing the values of their keys to determine their relative order in the hash_map.
hash_map Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.
hash_multimap Class	Used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

Requirements

Header: <hash_map>

Namespace: stdext

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<hash_map>` functions

Article • 12/07/2021

swap
swap (hash_map)

swap (hash_map)

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Exchanges the elements of two hash_maps.

```
C++
```

```
void swap(
    hash_map <Key, Type, Traits, Alloctor>& left,
    hash_map <Key, Type, Traits, Allocator>& right);
```

Parameters

right
The hash_map whose elements are to be exchanged with those of the map *left*.

left
The hash_map whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class hash_map to execute the member function `left.`swap*(right)*. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

swap

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Exchanges the elements of two hash_multimaps.

C++

```
void swap(
    hash_multimap <Key, Type, Traits, Alloctor>& left,
    hash_multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

right
The hash_multimap whose elements are to be exchanged with those of the map *left*.

left
The hash_multimap whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class hash_multimap to execute the member function `left.`swap*(right*`)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **template <class T> void swap(T&, T&)**, in the algorithm header file works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

See also

<hash_map>

`<hash_map>` operators

Article • 12/07/2021

operator!=
operator!= (multimap)
operator==
operator== (multimap)

operator!=

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Tests if the hash_map object on the left side of the operator is not equal to the hash_map object on the right side.

```
C++
```

```
bool operator!=(const hash_map <Key, Type, Traits, Allocator>& left, const
hash_map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_map`.

right
An object of type `hash_map`.

Return Value

`true` if the hash_maps are not equal; **`false`** if hash_maps are equal.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. Two hash_maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Members of the <hash_map> and <hash_set> header files in the stdext Namespace.

Example

C++

```
// hash_map_op_ne.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 != hm2 )
      cout << "The hash_maps hm1 and hm2 are not equal." << endl;
   else
      cout << "The hash_maps hm1 and hm2 are equal." << endl;

   if ( hm1 != hm3 )
      cout << "The hash_maps hm1 and hm3 are not equal." << endl;
   else
      cout << "The hash_maps hm1 and hm3 are equal." << endl;
}
```

Output

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

operator==

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Tests if the hash_map object on the left side of the operator is equal to the hash_map object on the right side.

```
C++
```

```
bool operator==(const hash_map <Key, Type, Traits, Allocator>& left, const
hash_map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_map`.

right
An object of type `hash_map`.

Return Value

`true` if the hash_map on the left side of the operator is equal to the hash_map on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between hash_map objects is based on a pairwise comparison of their elements. Two hash_maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// hash_map_op_eq.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;
```

```
   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 == hm2 )
      cout << "The hash_maps hm1 and hm2 are equal." << endl;
   else
      cout << "The hash_maps hm1 and hm2 are not equal." << endl;

   if ( hm1 == hm3 )
      cout << "The hash_maps hm1 and hm3 are equal." << endl;
   else
      cout << "The hash_maps hm1 and hm3 are not equal." << endl;
}
```

Output

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

operator!= (hash_multimap)

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Tests if the hash_multimap object on the left side of the operator is not equal to the hash_multimap object on the right side.

C++

```
bool operator!=(const hash_multimap <Key, Type, Traits, Allocator>& left,
const hash_multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_multimap`.

right
An object of type `hash_multimap`.

Return Value

`true` if the hash_multimaps are not equal; `false` if hash_multimaps are equal.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. Two hash_multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// hash_multimap_op_ne.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hm1.insert ( Int_Pair ( i, i ) );
      hm2.insert ( Int_Pair ( i, i * i ) );
      hm3.insert ( Int_Pair ( i, i ) );
   }

   if ( hm1 != hm2 )
      cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm2 are equal." << endl;

   if ( hm1 != hm3 )
      cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
}
```

Output

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

operator== (hash_multimap)

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Tests if the hash_multimap object on the left side of the operator is equal to the hash_multimap object on the right side.

```
C++
```

```
bool operator==(const hash_multimap <Key, Type, Traits, Allocator>& left,
const hash_multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_multimap`.

right
An object of type `hash_multimap`.

Return Value

`true` if the hash_multimap on the left side of the operator is equal to the hash_multimap on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between hash_multimap objects is based on a pairwise comparison of their elements. Two hash_multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// hash_multimap_op_eq.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap<int, int> hm1, hm2, hm3;
   int i;
   typedef pair<int, int> Int_Pair;

   for (i = 0; i < 3; i++)
   {
      hm1.insert(Int_Pair(i, i));
      hm2.insert(Int_Pair(i, i*i));
      hm3.insert(Int_Pair(i, i));
   }

   if ( hm1 == hm2 )
      cout << "The hash_multimaps hm1 and hm2 are equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm2 are not equal." << endl;

   if ( hm1 == hm3 )
      cout << "The hash_multimaps hm1 and hm3 are equal." << endl;
   else
      cout << "The hash_multimaps hm1 and hm3 are not equal." << endl;
}
```

Output

```
The hash_multimaps hm1 and hm2 are not equal.
The hash_multimaps hm1 and hm3 are equal.
```

See also

<hash_map>

hash_compare Class

Article • 10/20/2023

The class template describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default **Traits** parameter object to order and hash the elements they contain.

Syntax

C++

```
class hash_compare
{
   Traits comp;
   public:
   const size_t bucket_size = 4;
   const size_t min_buckets = 8;
   hash_compare();
   hash_compare(Traits pred);
   size_t operator()(const Key& key) const;
   bool operator()(
   const Key& key1,
   const Key& key2) const;
};
```

Remarks

Each hash associative container stores a hash traits object of type `Traits` (a template parameter). You can derive a class from a specialization of hash_compare to selectively override certain functions and objects, or you can supply your own version of this class if you meet certain minimum requirements. Specifically, for an object hash_comp of type `hash_compare<Key, Traits>`, the following behavior is required by the above containers:

- For all values `key` of type `Key`, the call **hash_comp**(`key`) serves as a hash function, which yields a distribution of values of type `size_t`. The function supplied by hash_compare returns `key`.

- For any value `key1` of type `Key` that precedes `key2` in the sequence and has the same hash value (value returned by the hash function), **hash_comp**(`key2`, `key1`) is false. The function must impose a total ordering on values of type `Key`. The function supplied by hash_compare returns *comp*(`key2`, `key1`) `,` where *comp* is a stored object of type `Traits` that you can specify when you construct the object

hash_comp. For the default `Traits` parameter type `less<Key>`, sort keys never decrease in value.

- The integer constant `bucket_size` specifies the mean number of elements per "bucket" (hash-table entry) that the container should try not to exceed. It must be greater than zero. The value supplied by hash_compare is 4.
- The integer constant `min_buckets` specifies the minimum number of buckets to maintain in the hash table. It must be a power of two and greater than zero. The value supplied by hash_compare is 8.

Example

See examples for hash_map::hash_map, hash_multimap::hash_multimap, hash_set::hash_set, and hash_multiset::hash_multiset, for examples of how to declare and use hash_compare.

Requirements

Header: <hash_map>

Namespace: stdext

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`hash_map` Class

Article • 10/17/2022

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Stores and retrieves data quickly from a collection in which each element is a pair that has a sort key whose value is unique and an associated data value.

Syntax

```
C++
```

```
template <class Key,
    class Type,
    class Traits=hash_compare<Key, less<Key>>,
    class Allocator=allocator<pair <const Key, Type>>>
class hash_map
```

Parameters

`Key`
The key data type to be stored in the `hash_map`.

`Type`
The element data type to be stored in the `hash_map`.

`Traits`
The type that includes two function objects, one of class compare able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and `hash_compare<Key, less<Key>>` is the default value.

`Allocator`
The type that represents the stored allocator object that encapsulates details about the `hash_map`'s allocation and deallocation of memory. This argument is optional, and the default value is `allocator<pair <const Key, Type>>`.

Remarks

The `hash_map` is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key.
- A pair associative container, because its element data values are distinct from its key values.
- A class template, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a `hash_map`, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The `hash_map` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of uniquely occurring keywords with associated string values providing, say, definitions. If, instead, the words had more than one correct

definition, so that keys weren't unique, then a `hash_multimap` would be the container of choice. If, on the other hand, just the list of words were being stored, then a `hash_set` would be the correct container. If multiple occurrences of the words were allowed, then a `hash_multiset` would be the appropriate container structure.

The `hash_map` orders the sequence it controls by calling a stored hash *`Traits`* object of class value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key>>`. Specifically, for all values *`Key`* of type *`Key`*, the call `Traits( Key )` yields a distribution of values of type `size_t`. For more information, see hash_compare.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they're equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate f(x y) is a function object that has two argument objects `x` and `y` and a return value of **`true`** or **`false`**. An ordering imposed on a `hash_map` is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects `x` and `y` are defined to be equivalent when both f(x, y) and f(y, x) are `false`. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You can't determine the current size of the hash table, so you can't in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the `hash_map` class is a bidirectional iterator, but the class member functions insert and hash_map have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to

the next iterator in the sequence. This is a minimal set of functionality, but it's enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the class member functions.

Constructors

Constructor	Description
hash_map	Constructs a `hash_map` that is empty or that is a copy of all or part of some other `hash_map`.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for the `hash_map` object.
const_iterator	A type that provides a bidirectional iterator that can read a **`const`** element in the `hash_map`.
const_pointer	A type that provides a pointer to a **`const`** element in a `hash_map`.
const_reference	A type that provides a reference to a **`const`** element stored in a `hash_map` for reading and performing **`const`** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **`const`** element in the `hash_map`.
difference_type	A signed integer type that can be used to represent the number of elements of a `hash_map` in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a `hash_map`.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `hash_map`.
key_type	A type describes the sort key object that constitutes each element of the `hash_map`.
mapped_type	A type that represents the data type stored in a `hash_map`.
pointer	A type that provides a pointer to an element in a `hash_map`.
reference	A type that provides a reference to an element stored in a `hash_map`.

Type name	Description
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `hash_map`.
size_type	An unsigned integer type that can represent the number of elements in a `hash_map`.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the `hash_map`.

Member functions

Member function	Description
at	Finds an element in a `hash_map` with a specified key value.
begin	Returns an iterator addressing the first element in the `hash_map`.
cbegin	Returns a const iterator addressing the first element in the `hash_map`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `hash_map`.
clear	Erases all the elements of a `hash_map`.
count	Returns the number of elements in a `hash_map` whose key matches a parameter-specified key.
crbegin	Returns a `const` iterator addressing the first element in a reversed `hash_map`.
crend	Returns a `const` iterator that addresses the location succeeding the last element in a reversed `hash_map`.
emplace	Inserts an element constructed in place into a `hash_map`.
emplace_hint	Inserts an element constructed in place into a `hash_map`, with a placement hint.
empty	Tests if a `hash_map` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `hash_map`.
equal_range	Returns a pair of iterators, respectively, to the first element in a `hash_map` with a key that is greater than a specified key and to the first element in the `hash_map` with a key that is equal to or greater than the key.

Member function	Description
erase	Removes an element or a range of elements in a `hash_map` from specified positions
find	Returns an iterator addressing the location of an element in a `hash_map` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `hash_map`.
insert	Inserts an element or a range of elements into a `hash_map`.
key_comp	Returns an iterator to the first element in a `hash_map` with a key value that is equal to or greater than that of a specified key.
lower_bound	Returns an iterator to the first element in a `hash_map` with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the `hash_map`.
rbegin	Returns an iterator addressing the first element in a reversed `hash_map`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `hash_map`.
size	Returns the number of elements in the `hash_map`.
swap	Exchanges the elements of two `hash_map`s.
upper_bound	Returns an iterator to the first element in a `hash_map` that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a `hash_map`.

Operators

Operator	Description
operator[]	Inserts an element into a `hash_map` with a specified key value.
hash_map::operator=	Replaces the elements of a `hash_map` with a copy of another `hash_map`.

Requirements

Header: `<hash_map>`

Namespace: `stdext`

`hash_map::allocator_type`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that represents the allocator class for the hash_map object.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::allocator_type allocator_type;
```

Example

See example for get_allocator for an example using `allocator_type`.

`hash_map::at`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Finds an element in a `hash_map` with a specified key value.

```
C++
```

```
Type& at(const Key& key);

const Type& at(const Key& key) const;
```

Parameters

`key`
The key value of the element that is to be found.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value isn't found, then the function throws an object of class out_of_range Class.

Example

C++

```
// hash_map_at.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_map <int, int> hm1;

   // Insert data values
   hm1.insert ( cInt2Int ( 1, 10 ) );
   hm1.insert ( cInt2Int ( 2, 20 ) );
   hm1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The values of the mapped elements are:";
   for ( int i = 1 ; i <= hm1.size() ; i++ )
      cout << " " << hm1.at(i);
   cout << "." << endl;
}
```

hash_map::begin

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator addressing the first element in the `hash_map`.

C++

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the `hash_map` or the location succeeding an empty `hash_map`.

Example

C++

```
// hash_map_begin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 0, 0 ) );
   hm1.insert ( Int_Pair ( 1, 1 ) );
   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.begin ( );
   cout << "The first element of hm1 is "
        << hm1_cIter -> first << "." << endl;

   hm1_Iter = hm1.begin ( );
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.begin ( );
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.begin( );
   cout << "The first element of hm1 is now "
        << hm1_cIter -> first << "." << endl;
}
```

Output

```
The first element of hm1 is 0.
The first element of hm1 is now 1.
```

`hash_map::cbegin`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns a const iterator addressing the first element in the `hash_map`.

C++

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the hash_map or the location succeeding an empty `hash_map`.

Example

C++

```
// hash_map_cbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.cbegin ( );
   cout << "The first element of hm1 is "
```

```
        << hm1_cIter -> first << "." << endl;
    }
```

Output

```
The first element of hm1 is 2.
```

hash_map::cend

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns a const iterator that addresses the location succeeding the last element in a `hash_map`.

C++

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a hash_map. If the `hash_map` is empty, then `hash_map::cend == hash_map::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_map`.

The value returned by `cend` shouldn't be dereferenced.

Example

C++

```
// hash_map_cend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.cend( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;
   }
```

Output

```
The value of last element of hm1 is 30.
```

`hash_map::clear`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Erases all the elements of a `hash_map`.

C++

```
void clear();
```

Remarks

Example

The following example demonstrates the use of the `hash_map::clear` member function.

C++

```
// hash_map_clear.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>
```

```
int main( )
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_map is initially "
         << i << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_map after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the hash_map is initially 2.
The size of the hash_map after clearing is 0.
```

`hash_map::const_iterator`

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

A type that provides a bidirectional iterator that can read a `const` element in the `hash_map`.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

The `const_iterator` defined by `hash_map` points to elements that are objects of value_type, that is of type `pair< const Key, Type >`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_iterator` `cIter` pointing to an element in a `hash_map`, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See example for begin for an example using `const_iterator`.

`hash_map::const_pointer`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a pointer to a **`const`** element in a `hash_map`.

```
C++
```

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `hash_map` object.

`hash_map::const_reference`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a reference to a `const` element stored in a `hash_map` for reading and performing `const` operations.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_reference const_reference;
```

Remarks

Example

C++

```
// hash_map_const_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map<int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of the first element in the hash_map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin( ) -> second );

   cout << "The data value of the first element in the hash_map is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of the first element in the hash_map is 1.
The data value of the first element in the hash_map is 10.
```

`hash_map::const_reverse_iterator`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a bidirectional iterator that can read any `const` element in the `hash_map`.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_reverse)iterator const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the `hash_map` in reverse.

The `const_reverse_iterator` defined by `hash_map` points to elements that are objects of value_type, that is of type `pair< const Key, Type >`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator` `crIter` pointing to an element in a `hash_map`, use the `->` operator.

To access the value of the key for the element, use `crIter->first`, which is equivalent to `(*crIter).first`. To access the value of the mapped datum for the element, use `crIter->second`, which is equivalent to `(*crIter).first`.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

`hash_map::count`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns the number of elements in a `hash_map` whose key matches a parameter-specified key.

```
C++
```

```
size_type count(const Key& key) const;
```

Parameters

`key`
The key value of the elements to be matched from the `hash_map`.

Return Value

1 if the `hash_map` contains an element whose sort key matches the parameter key; 0 if the `hash_map` doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements *`x`* in the range

lower_bound(key), upper_bound(key)

which is 0 or 1 in the case of `hash_map`, which is a unique associative container.

Example

The following example demonstrates the use of the `hash_map::count` member function.

```
C++
```

```
// hash_map_count.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>
```

```cpp
int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair (1, 1));
    hm1.insert(Int_Pair (2, 1));
    hm1.insert(Int_Pair (1, 4));
    hm1.insert(Int_Pair (2, 1));

    // Keys must be unique in hash_map, so duplicates are ignored
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hm1 with a sort key of 1 is: 1.
The number of elements in hm1 with a sort key of 2 is: 1.
The number of elements in hm1 with a sort key of 3 is: 0.
```

`hash_map::crbegin`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns a `const` iterator addressing the first element in a reversed `hash_map`.

C++

```cpp
const_reverse_iterator crbegin() const;
```

Return Value

A `const` reverse bidirectional iterator addressing the first element in a reversed hash_map or addressing what had been the last element in the unreversed `hash_map`.

Remarks

`crbegin` is used with a reversed `hash_map` just as begin is used with a `hash_map`.

With the return value of `crbegin`, the `hash_map` object can't be modified.

`crbegin` can be used to iterate through a `hash_map` backwards.

Example

C++

```
// hash_map_crbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_crIter = hm1.crbegin( );
   cout << "The first element of the reversed hash_map hm1 is "
        << hm1_crIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_map hm1 is 3.
```

`hash_map::crend`

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns a `const` iterator that addresses the location succeeding the last element in a reversed `hash_map`.

```
C++
```

```
const_reverse_iterator crend() const;
```

Return Value

A `const` reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_map (the location that had preceded the first element in the unreversed `hash_map`).

Remarks

`crend` is used with a reversed `hash_map` just as hash_map::end is used with a `hash_map`.

With the return value of `crend`, the `hash_map` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `hash_map`.

The value returned by `crend` shouldn't be dereferenced.

Example

```
C++
```

```
// hash_map_crend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
```

```
   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_crIter = hm1.crend( );
   hm1_crIter--;
   cout << "The last element of the reversed hash_map hm1 is "
        << hm1_crIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_map hm1 is 3.
```

hash_map::difference_type

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

A signed integer type that can be used to represent the number of elements of a `hash_map` in a range between elements pointed to by iterators.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::difference_type difference_type;
```

Example

C++

```
// hash_map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   typedef pair <int, int> Int_Pair;
```

```
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );

   // The following won't insert, because map keys are unique
   hm1.insert ( Int_Pair ( 2, 30 ) );

   hash_map <int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
   hm1_bIter = hm1.begin( );
   hm1_eIter = hm1.end( );

   // Count the number of elements in a hash_map
   hash_map <int, int>::difference_type  df_count = 0;
   hm1_Iter = hm1.begin( );
   while ( hm1_Iter != hm1_eIter)
   {
      df_count++;
      hm1_Iter++;
   }

   cout << "The number of elements in the hash_map hm1 is: "
        << df_count << "." << endl;

   cout  << "The keys of the mapped elements are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
         hm1_Iter++)
      cout << " " << hm1_Iter-> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( hm1_Iter= hm1.begin( ) ; hm1_Iter!= hm1.end( ) ;
         hm1_Iter++)
      cout << " " << hm1_Iter-> second;
   cout << "." << endl;
}
```

Output

```
The number of elements in the hash_map hm1 is: 3.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 20.
```

hash_map::emplace

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

Inserts an element constructed in place into a `hash_map`.

```
C++
```

```
template <class ValTy>
pair <iterator, bool>
emplace(
    ValTy&& val);
```

Parameters

val
The value used to move construct an element to be inserted into the hash_map unless the `hash_map` already contains that element (or, more generally, an element whose key is equivalently ordered).

Return Value

The `emplace` member function returns a pair whose `bool` component returns `true` if an insertion was made and `false` if the `hash_map` already contained an element whose key had an equivalent value in the ordering, and whose `iterator` component returns the address where a new element was inserted or where the element was already located.

To access the `iterator` component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **`bool`** component of a pair `pr` returned by this member function, use `pr.second`, and to dereference it, use `*(pr.second)`.

Remarks

The hash_map::value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

```
C++
```

```
// hash_map_emplace.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
```

```
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(move(is1));
    cout << "After the emplace insertion, hm1 contains:" << endl
      << " " << hm1.begin()->first
      << " => " << hm1.begin()->second
      << endl;
}
```

Output

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_map::emplace_hint

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Inserts an element constructed in place into a `hash_map`, with a placement hint.

C++

```
template <class ValTy>
iterator emplace_hint(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

`val`

The value used to move construct an element to be inserted into the hash_map unless the `hash_map` already contains that element (or, more generally, an element whose key is equivalently ordered).

`c`

A hint regarding the place to start searching for the correct point of insertion.

Return Value

The hash_multimap::emplace member function returns an iterator that points to the position where the new element was inserted into the `hash_map`, or where the existing element with equivalent ordering is located.

Remarks

The hash_map::value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `_Where`.

Example

C++

```
// hash_map_emplace_hint.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(hm1.begin(), move(is1));
    cout << "After the emplace, hm1 contains:" << endl
      << " " << hm1.begin()->first
      << " => " << hm1.begin()->second
      << endl;
}
```

Output

```
After the emplace insertion, hm1 contains:
1 => a
```

`hash_map::empty`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Tests if a `hash_map` is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the `hash_map` is empty; **`false`** if the `hash_map` is nonempty.

Remarks

Example

C++

```
// hash_map_empty.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2;

   typedef pair <int, int> Int_Pair;
   hm1.insert ( Int_Pair ( 1, 1 ) );

   if ( hm1.empty( ) )
      cout << "The hash_map hm1 is empty." << endl;
   else
      cout << "The hash_map hm1 is not empty." << endl;

   if ( hm2.empty( ) )
      cout << "The hash_map hm2 is empty." << endl;
```

```
    else
        cout << "The hash_map hm2 is not empty." << endl;
}
```

Output

```
The hash_map hm1 is not empty.
The hash_map hm2 is empty.
```

hash_map::end

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator that addresses the location succeeding the last element in a `hash_map`.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a `hash_map`. If the `hash_map` is empty, then `hash_map::end == hash_map::begin`.

Remarks

`end` is used to test whether an iterator has reached the end of its `hash_map`.

The value returned by `end` shouldn't be dereferenced.

Example

C++

```
// hash_map_end.cpp
// compile with: /EHsc
```

```
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;

   hm1_Iter = hm1.end( );
   hm1_Iter--;
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.end ( );
   // hm1_cIter--;
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is now "
        << hm1_cIter -> second << "." << endl;
}
```

Output

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

hash_map::equal_range

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

Returns a pair of iterators respectively to the first element in a `hash_map` with a key that is greater than a specified key and to the first element in the `hash_map` with a key that is equal to or greater than the key.

```
C++
```

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key value to be compared with the sort key of an element from the `hash_map` being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Remarks

Example

```
C++
```

```
// hash_map_equal_range.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_map <int, int> IntMap;
   IntMap hm1;
```

```
   hash_map <int, int> :: const_iterator hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
   p1 = hm1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the hash_map hm1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the hash_map hm1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   hm1_RcIter = hm1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << hm1_RcIter -> second << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 2 )." << endl;

   p2 = hm1.equal_range( 4 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of hash_map hm1 with a key >= 40 is: "
           << p1.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2 in the hash_map hm1 is: 20.
The upper bound of the element with a key of 2 in the hash_map hm1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 2 ).
The hash_map hm1 doesn't have an element with a key less than 40.
```

hash_map::erase

ⓘ Note

This API is obsolete. The alternative is **unordered_map Class**.

Removes an element or a range of elements in a `hash_map` from specified positions or removes elements that match a specified key.

```
C++
```

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

`_Where`
Position of the element to be removed from the `hash_map`.

`first`
Position of the first element removed from the `hash_map`.

`last`
Position just beyond the last element removed from the `hash_map`.

`key`
The key value of the elements to be removed from the `hash_map`.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the `hash_map` if no such element exists.

For the third member function, returns the number of elements that have been removed from the `hash_map`.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the `hash_map::erase` member function.

C++

```
// hash_map_erase.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2, hm3;
    hash_map<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_map<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;

    for (i = 1; i < 5; i++)
    {
        hm1.insert(Int_Pair (i, i));
        hm2.insert(Int_Pair (i, i*i));
        hm3.insert(Int_Pair (i, i-1));
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hm1.begin();
    hm1.erase(Iter1);

    cout << "After the 2nd element is deleted, the hash_map hm1 is:";
    for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [ first,  last)
    Iter1 = ++hm2.begin();
    Iter2 = --hm2.end();
    hm2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_map hm2 is:";
    for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given  key
    n = hm3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the hash_map hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
```

```
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hm3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hm3.begin();
    hm3.erase(Iter1);

    cout << "After another element with a key equal to that"
         << endl;
    cout  << "of the 2nd element is deleted, "
          << "the hash_map hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_map hm1 is: 1 3 4.
After the middle two elements are deleted, the hash_map hm2 is: 1 16.
After the element with a key of 2 is deleted,
the hash_map hm3 is: 0 2 3.
The number of elements removed from hm3 is: 1.
After another element with a key equal to that
of the 2nd element is deleted, the hash_map hm3 is: 0 3.
```

hash_map::find

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator addressing the location of an element in a `hash_map` that has a key equivalent to a specified key.

C++

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key

The key value to be matched by the sort key of an element from the `hash_map` being searched.

Return Value

An iterator that addresses the location of an element with a specified key, or the location succeeding the last element in the `hash_map` if no match is found for the key.

Remarks

`find` returns an iterator that addresses an element in the `hash_map` whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a const_iterator, the `hash_map` object can't be modified. If the return value of `find` is assigned to an iterator, the `hash_map` object can be modified

Example

C++

```
// hash_map_find.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.find( 2 );
   cout << "The element of hash_map hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.find( 4 );
```

```
   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_map can be found
   // using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.end( );
   hm1_AcIter--;
   hm1_RcIter = hm1.find( hm1_AcIter -> first );
   cout << "The element of hm1 with a key matching "
        << "that of the last element is: "
        << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.
```

hash_map::get_allocator

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

Returns a copy of the allocator object used to construct the `hash_map`.

C++

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the `hash_map`.

Remarks

Allocators for the `hash_map` class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// hash_map_get_allocator.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int>::allocator_type hm1_Alloc;
   hash_map <int, int>::allocator_type hm2_Alloc;
   hash_map <int, double>::allocator_type hm3_Alloc;
   hash_map <int, int>::allocator_type hm4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   hash_map <int, int> hm1;
   hash_map <int, int> hm2;
   hash_map <int, double> hm3;

   hm1_Alloc = hm1.get_allocator( );
   hm2_Alloc = hm2.get_allocator( );
   hm3_Alloc = hm3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hm2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hm3.max_size( ) <<  "." << endl;

   // The following line creates a hash_map hm4
   // with the allocator of hash_map hm1.
   hash_map <int, int> hm4( less<int>( ), hm1_Alloc );

   hm4_Alloc = hm4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( hm1_Alloc == hm4_Alloc )
   {
```

```
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

hash_map::hash_map

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Constructs a `hash_map` that is empty or is a copy of all or part of some other `hash_map`.

C++

```
hash_map();

explicit hash_map(
    const Traits& Comp);

hash_map(
    const Traits& Comp,
    const Allocator& Al);

hash_map(
    const hash_map& Right);

hash_map(
    hash_map&& Right);

hash_map(
    initializer_list<Type> IList);hash_map(initializer_list<Type> IList,
    const key_compare& Comp);

hash_map(
    initializer_list<Type> IList,
    const key_compare& Comp,
    const allocator_type& Al);

template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last);
```

```
template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
hash_map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al
```

Parameters

Al

The storage allocator class to be used for this `hash_map` object, which defaults to `Allocator`.

Comp

The comparison function of type const `Traits` used to order the elements in the `hash_map`, which defaults to `hash_compare`.

Right

The `hash_map` of which the constructed map is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The `initializer_list`

Remarks

All constructors store a type of allocator object that manages memory storage for the `hash_map` and can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their `hash_map`.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_map` and that can later be returned by calling key_comp.

The first three constructors specify an empty initial `hash_map`, in addition, the second specifies the type of comparison function (*`Comp`*) to be used in establishing the order of the elements and the third explicitly specifies the allocator type (*`Al`*) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the `hash_map` *`Right`*.

The next three constructors copy the range `[First, Last)` of a `hash_map` with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

The last constructor moves the `hash_map` *`Right`*.

`hash_map::insert`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Inserts an element or a range of elements into a `hash_map`.

C++

```
pair <iterator, bool> insert(
    const value_type& val);

iterator insert(
    const_iterator _Where,
    const value_type& val);

template <class InputIterator>
void insert(
    InputIterator first,
    InputIterator last);

template <class ValTy>
pair <iterator, bool>
insert(
    ValTy&& val);

template <class ValTy>
iterator insert(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

val

The value of an element to be inserted into the hash_map unless the hash_map already contains that element (or, more generally, an element whose key is equivalently ordered).

_Where

A hint regarding the place to start searching for the correct point of insertion.

first

The position of the first element to be copied from a hash_map.

last

The position just beyond the last element to be copied from a hash_map.

Return Value

The first insert member function returns a pair whose bool component returns true if an insertion was made and false if the hash_map already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair pr returned by this member function, use pr.first, and to dereference it, use (pr.first). To access the **bool** component of a pair pr returned by this member function, use pr.second, and to dereference it, use \(pr.second).

The second insert member function, the hint version, returns an iterator that points to the position where the new element was inserted into the hash_map.

The last two insert member functions behave the same as the first two, except that they move construct the inserted value.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows `_Where`.

The third member function inserts the sequence of element values into a `hash_map` corresponding to each element addressed by an iterator of in the range `*[First, Last)*` of a specified set.

Example

C++

```
// hash_map_insert.cpp
// compile with: /EHsc
#include<hash_map>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int>::iterator hm1_pIter, hm2_pIter;

    hash_map<int, int> hm1, hm2;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(2, 20));
    hm1.insert(Int_Pair(3, 30));
    hm1.insert(Int_Pair(4, 40));

    cout << "The original elements (Key => Value) of hm1 are:";
    for (hm1_pIter = hm1.begin(); hm1_pIter != hm1.end(); hm1_pIter++)
        cout << endl << " " << hm1_pIter -> first << " => "
             << hm1_pIter->second;
    cout << endl;

    pair< hash_map<int,int>::iterator, bool > pr;
    pr = hm1.insert(Int_Pair(1, 10));

    if (pr.second == true)
    {
        cout << "The element 10 was inserted in hm1 successfully."
            << endl;
    }
    else
    {
        cout << "The element 10 already exists in hm1\n"
            << "with a key value of "
            << "((pr.first) -> first) = " << (pr.first)->first
            << "." << endl;
```

```
    }

    // The hint version of insert
    hm1.insert(--hm1.end(), Int_Pair(5, 50));

    cout << "After the insertions, the elements of hm1 are:";
    for (hm1_pIter = hm1.begin(); hm1_pIter != hm1.end(); hm1_pIter++)
        cout << endl << hm1_pIter -> first << " => "
             << hm1_pIter->second;
    cout << endl;

    hm2.insert(Int_Pair(10, 100));

    // The templatized version inserting a range
    hm2.insert( ++hm1.begin(), --hm1.end() );

    cout << "After the insertions, the elements of hm2 are:";
    for (hm2_pIter = hm2.begin(); hm2_pIter != hm2.end(); hm2_pIter++)
        cout << endl << hm2_pIter -> first << " => "
             << hm2_pIter->second;
    cout << endl;

    // The templatized versions move constructing elements
    hash_map<int, string> hm3, hm4;
    pair<int, string> is1(1, "a"), is2(2, "b");

    hm3.insert(move(is1));
    cout << "After the move insertion, hm3 contains:" << endl
      << hm3.begin()->first
      << " => " << hm3.begin()->second
      << endl;

    hm4.insert(hm4.begin(), move(is2));
    cout << "After the move insertion, hm4 contains:" << endl
      << hm4.begin()->first
      << " => " << hm4.begin()->second
      << endl;
}
```

Output

```
The original elements (Key => Value) of hm1 are:
1 => 10
2 => 20
3 => 30
4 => 40
The element 10 already exists in hm1
with a key value of ((pr.first) -> first) = 1.
After the insertions, the elements of hm1 are:
1 => 10
2 => 20
3 => 30
4 => 40
```

```
5 => 50
After the insertions, the elements of hm2 are:
2 => 20
10 => 100
3 => 30
4 => 40
After the move insertion, hm3 contains:
1 => a
After the move insertion, hm4 contains:
2 => b
```

`hash_map::iterator`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a bidirectional iterator that can read or modify any element in a `hash_map`.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::iterator iterator;
```

Remarks

The `iterator` defined by `hash_map` points to elements that are objects of value_type, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an iterator named `Iter` pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).second`.

A type `iterator` can be used to modify the value of an element.

Example

See example for begin for an example of how to declare and use the `iterator`.

`hash_map::key_comp`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Retrieves a copy of the comparison object used to order keys in a `hash_map`.

C++

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a `hash_map` uses to order its elements.

Remarks

The stored object defines the member function

`bool operator( const Key& left, const Key&right );`

that returns **`true`** if `left` precedes and isn't equal to `right` in the sort order.

Example

C++

```
// hash_map_key_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_map <int, int, hash_compare<int, less<int> > > hm1;
   hash_map <int, int, hash_compare<int, less<int> > >::key_compare
      kc1 = hm1.key_comp( ) ;
```

```
   // Operator stored in kc1 tests order & returns bool value
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true,"
           << "\n where kc1 is the function object of hm1"
           << " of type key_compare." << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false"
           << "\n where kc1 is the function object of hm1"
           << " of type key_compare." << endl;
   }

   hash_map <int, int, hash_compare<int, greater<int> > > hm2;
   hash_map <int, int, hash_compare<int, greater<int> > >
      ::key_compare kc2 = hm2.key_comp( );

   // Operator stored in kc2 tests order & returns bool value
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true,"
           << "\n where kc2 is the function object of hm2"
           << " of type key_compare." << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false,"
           << "\n where kc2 is the function object of hm2"
           << " of type key_compare." << endl;
   }
}
```

hash_map::key_compare

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the map.

C++

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the hash_map Class topic.

Example

See example for key_comp for an example of how to declare and use `key_compare`.

`hash_map::key_type`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type describes the sort key object that constitutes each element of the `hash_map`.

```
C++
```

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the hash_map Class topic.

Example

See example for value_type for an example of how to declare and use `key_type`.

`hash_map::lower_bound`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator to the first element in a `hash_map` with a key value that is equal to or greater than that of a specified key.

C++

```
iterator lower_bound(const Key& key);

const_iterator lower_bound(const Key& key) const;
```

Parameters

key
The argument key value to be compared with the sort key of an element from the `hash_map` being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a `hash_map` that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the `hash_map` if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the `hash_map` object can't be modified. If the return value of `lower_bound` is assigned to a `iterator`, the `hash_map` object can be modified.

Remarks

Example

C++

```
// hash_map_lower_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;
```

```
   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.lower_bound( 2 );
   cout << "The first element of hash_map hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1. lower_bound ( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // An element at a specific location in the hash_map can be
   // found using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.end( );
   hm1_AcIter--;
   hm1_RcIter = hm1. lower_bound ( hm1_AcIter -> first );
   cout << "The element of hm1 with a key matching "
        << "that of the last element is: "
        << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of hash_map hm1 with a key of 2 is: 20.
The hash_map hm1 doesn't have an element with a key of 4.
The element of hm1 with a key matching that of the last element is: 30.
```

hash_map::mapped_type

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

A type that represents the data type stored in a `hash_map`.

C++

```
typedef Type mapped_type;
```

Remarks

The type `mapped_type` is a synonym for the template parameter `Type`.

For more information on `Type` see the hash_map Class topic.

Example

See example for value_type for an example of how to declare and use `key_type`.

`hash_map::max_size`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns the maximum length of the `hash_map`.

```
C++
```

```
size_type max_size() const;
```

Return Value

The maximum possible length of the `hash_map`.

Remarks

Example

```
C++
```

```
// hash_map_max_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
```

```
    hash_map <int, int> :: size_type i;

    i = hm1.max_size( );
    cout << "The maximum possible length "
         << "of the hash_map is " << i << "."
         << endl << "(Magnitude is machine specific.)";
}
```

`hash_map::operator[]`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Inserts an element into a `hash_map` with a specified key value.

```
C++

Type& operator[](const Key& key);

Type& operator[](Key&& key);
```

Parameters

`key`
The key value of the element that is to be inserted.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value isn't found, then it's inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a `hash_map m` using

`m[ key] = DataValue`;

where DataValue is the value of the `mapped_type` of the element with a key value of *`key`*.

When using `operator[]` to insert elements, the returned reference doesn't indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

Example

C++

```
// hash_map_op_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_map <int, int> hm1;
   hash_map <int, int> :: iterator pIter;

   // Insert a data value of 10 with a key of 1
   // into a hash_map using the operator[] member function
   hm1[ 1 ] = 10;

   // Compare other ways to insert objects into a hash_map
   hm1.insert ( hash_map <int, int> :: value_type ( 2, 20 ) );
   hm1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

   // If the key already exists, operator[]
   // changes the value of the datum in the element
   hm1[ 2 ] = 40;

   // operator[] will also insert the value of the data
   // type's default constructor if the value is unspecified
   hm1[5];

   cout  << "The keys of the mapped elements are now:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
```

```
    cout << "." << endl;

    cout  << "The values of the mapped elements are now:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;

    // operator[] will also insert by moving a key
    hash_map <string, int> hm2;
    string str("a");
    hm2[move(str)] = 1;
    cout << "The moved key is " << hm2.begin()->first
       << ", with value " << hm2.begin()->second << endl;
}
```

`hash_map::operator=`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Replaces the elements of the `hash_map` with a copy of another `hash_map`.

C++

```
hash_map& operator=(const hash_map& right);

hash_map& operator=(hash_map&& right);
```

Parameters

`right`
The right Class being copied into the `hash_map`.

Remarks

After erasing any existing elements in a `hash_map`, `operator=` either copies or moves the contents of *`right`* into the `hash_map`.

Example

C++

```cpp
// hash_map_operator_as.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map<int, int> v1, v2, v3;
   hash_map<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
}
```

hash_map::pointer

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

A type that provides a pointer to an element in a `hash_map`.

C++

```cpp
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `hash_map` object.

`hash_map::rbegin`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator addressing the first element in a reversed `hash_map`.

```
C++
```

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed `hash_map` or addressing what had been the last element in the unreversed `hash_map`.

Remarks

`rbegin` is used with a reversed `hash_map` just as begin is used with a `hash_map`.

If the return value of `rbegin` is assigned to a const_reverse_iterator, then the `hash_map` object can't be modified. If the return value of `rbegin` is assigned to a reverse_iterator, then the `hash_map` object can be modified.

`rbegin` can be used to iterate through a `hash_map` backwards.

Example

```
C++
```

```
// hash_map_rbegin.cpp
// compile with: /EHsc
```

```
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: reverse_iterator hm1_rIter;
   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rbegin( );
   cout << "The first element of the reversed hash_map hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_map in a forward order
   cout << "The hash_map is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_map in a reverse order
   cout << "The reversed hash_map is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;

   // A hash_map element can be erased by dereferencing to its key
   hm1_rIter = hm1.rbegin( );
   hm1.erase ( hm1_rIter -> first );

   hm1_rIter = hm1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed hash_map is "
        << hm1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_map hm1 is 3.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the first element in the reversed hash_map is 2.
```

hash_map::reference

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that provides a reference to an element stored in a `hash_map`.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::reference reference;
```

Remarks

Example

C++

```
// hash_map_reference.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
```

```
    int &Ref2 = ( hm1.begin( ) -> second );

    cout << "The data value of first element in the hash_map is "
         << Ref2 << "." << endl;

    // The non-const_reference can be used to modify the
    // data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The modified data value of first element is 15.
```

hash_map::rend

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator that addresses the location succeeding the last element in a reversed `hash_map`.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed `hash_map` (the location that had preceded the first element in the unreversed `hash_map`).

Remarks

`rend` is used with a reversed `hash_map` just as end is used with a `hash_map`.

If the return value of `rend` is assigned to a const_reverse_iterator, then the `hash_map` object can't be modified. If the return value of `rend` is assigned to a reverse_iterator, then the `hash_map` object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its `hash_map`.

The value returned by `rend` shouldn't be dereferenced.

Example

C++

```
// hash_map_rend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;

   hash_map <int, int> :: iterator hm1_Iter;
   hash_map <int, int> :: reverse_iterator hm1_rIter;
   hash_map <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "The last element of the reversed hash_map hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_map in a forward order
   cout << "The hash_map is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( );
   hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_map in a reverse order
   cout << "The reversed hash_map is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( );
      hm1_rIter++)
```

```
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;

   // A hash_map element can be erased by dereferencing to its key
   hm1_rIter = --hm1.rend( );
   hm1.erase ( hm1_rIter -> first );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "After the erasure, the last element "
        << "in the reversed hash_map is "
        << hm1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_map hm1 is 1.
The hash_map is: 1 2 3 .
The reversed hash_map is: 3 2 1 .
After the erasure, the last element in the reversed hash_map is 2.
```

`hash_map::reverse_iterator`

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

A type that provides a bidirectional iterator that can read or modify an element in a reversed `hash_map`.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::reverse_iterator reverse_iterator;
```

Remarks

A type `reverse_iterator` can't modify the value of an element and is used to iterate through the `hash_map` in reverse.

The `reverse_iterator` defined by `hash_map` points to elements that are objects of value_type, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `reverse_iterator` `rIter` pointing to an element in a `hash_map`, use the `->` operator.

To access the value of the key for the element, use `rIter->first`, which is equivalent to `(*rIter).first`. To access the value of the mapped datum for the element, use `rIter->second`, which is equivalent to `(*rIter).first`.

Example

See example for rbegin for an example of how to declare and use `reverse_iterator`.

`hash_map::size`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns the number of elements in the `hash_map`.

C++

```
size_type size() const;
```

Return Value

The current length of the `hash_map`.

Remarks

Example

The following example demonstrates the use of the `hash_map::size` member function.

C++

```
// hash_map_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
```

```
{
    using namespace std;
    using namespace stdext;
    hash_map<int, int> hm1, hm2;
    hash_map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_map length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_map length is now " << i << "." << endl;
}
```

Output

```
The hash_map length is 1.
The hash_map length is now 2.
```

hash_map::size_type

> ⓘ Note
>
> This API is obsolete. The alternative is unordered_map Class.

An unsigned integer type that can represent the number of elements in a `hash_map`.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See example for size for an example of how to declare and use `size_type`

hash_map::swap

> **Note**
>
> This API is obsolete. The alternative is **unordered_map Class**.

Exchanges the elements of two `hash_map`s.

```
C++
```

```
void swap(hash_map& right);
```

Parameters

`right`
The argument `hash_map` providing the elements to be swapped with the target `hash_map`.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two `hash_map`s whose elements are being exchanged.

Example

```
C++
```

```
// hash_map_swap.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1, hm2, hm3;
   hash_map <int, int>::iterator hm1_Iter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm2.insert ( Int_Pair ( 10, 100 ) );
   hm2.insert ( Int_Pair ( 20, 200 ) );
   hm3.insert ( Int_Pair ( 30, 300 ) );
```

```
   cout << "The original hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   // hm2 is said to be the argument hash_map;
   // hm1 is said to be the target hash_map
   hm1.swap( hm2 );

   cout << "After swapping with hm2, hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hm1, hm3 );

   cout << "After swapping with hm3, hash_map hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original hash_map hm1 is: 10 20 30.
After swapping with hm2, hash_map hm1 is: 100 200.
After swapping with hm3, hash_map hm1 is: 300.
```

hash_map::upper_bound

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns an iterator to the first element in a `hash_map` that with a key having a value that is greater than that of a specified key.

C++

```
iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;
```

Parameters

key
The argument key value to be compared with the sort key value of an element from the `hash_map` being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a `hash_map` that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the `hash_map` if no match is found for the key.

If the return value is assigned to a `const_iterator`, the `hash_map` object can't be modified. If the return value is assigned to an `iterator`, the `hash_map` object can be modified.

Remarks

Example

C++

```
// hash_map_upper_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_map <int, int> hm1;
   hash_map <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.upper_bound( 2 );
   cout << "The first element of hash_map hm1 with a key "
        << "greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end is returned
   hm1_RcIter = hm1. upper_bound ( 4 );
```

```
   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_map hm1 doesn't have an element "
           << "with a key greater than 4." << endl;
   else
      cout << "The element of hash_map hm1 with a key > 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_map can be found
   // using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.begin( );
   hm1_RcIter = hm1. upper_bound ( hm1_AcIter -> first );
   cout << "The 1st element of hm1 with a key greater than that\n"
        << "of the initial element of hm1 is: "
        << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of hash_map hm1 with a key greater than 2 is: 30.
The hash_map hm1 doesn't have an element with a key greater than 4.
The 1st element of hm1 with a key greater than that
of the initial element of hm1 is: 20.
```

`hash_map::value_comp`

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

Returns a function object that determines the order of elements in a `hash_map` by comparing their key values.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a `hash_map` uses to order its elements.

Remarks

For a `hash_map` *`m`*, if two elements *`e1`* (*`k1`*, *`d1`*) and *`e2`* (*`k2`*, *`d2`*) are objects of type value_type, where *`k1`* and *`k2`* are their keys of type key_type and *`d1`* and *`d2`* are their data of type mapped_type, then `m.value_comp()(e1, e2)` is equivalent to `m.key_comp()(k1, k2)`. A stored object defines the member function

```
bool operator(value_type& left, value_type& right);
```

which returns **`true`** if the key value of `left` precedes and isn't equal to the key value of `right` in the sort order.

Example

C++

```
// hash_map_value_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_map <int, int, hash_compare<int, less<int> > > hm1;
   hash_map <int, int, hash_compare<int, less<int> > >
   ::value_compare vc1 = hm1.value_comp( );
   pair< hash_map<int,int>::iterator, bool > pr1, pr2;

   pr1= hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );
   pr2= hm1.insert ( hash_map <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *pr1.first, *pr2.first ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1 ( *pr2.first, *pr1.first ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
```

```
      cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
           << endl;
   }
}
```

hash_map::value_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_map Class**.

A type that represents the type of object stored in a `hash_map`.

C++

```
typedef pair<const Key, Type> value_type;
```

Remarks

`value_type` is declared to be `pair<const key_type, mapped_type>` and not `pair<key_type, mapped_type>` because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example

C++

```
// hash_map_value_type.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_map <int, int> hm1;
   hash_map <int, int> :: key_type key1;
   hash_map <int, int> :: mapped_type mapped1;
   hash_map <int, int> :: value_type value1;
   hash_map <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
```

```
    // explicitly to avoid implicit type conversion
    hm1.insert ( hash_map <int, int> :: value_type ( 1, 10 ) );

    // Compare other ways to insert objects into a hash_map
    hm1.insert ( cInt2Int ( 2, 20 ) );
    hm1[ 3 ] = 30;

    // Initializing key1 and mapped1
    key1 = ( hm1.begin( ) -> first );
    mapped1 = ( hm1.begin( ) -> second );

    cout << "The key of first element in the hash_map is "
         << key1 << "." << endl;

    cout << "The data value of first element in the hash_map is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout  << "The keys of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are:";
    for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
The key of first element in the hash_map is 1.
The data value of first element in the hash_map is 10.
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

hash_multimap Class

Article • 08/03/2021

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

The container class hash_multimap is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which each element is a pair that has a sort key whose value need not be unique and an associated data value.

Syntax

C++

```
template <class Key,
    class Type,
    class Traits=hash_compare <Key, less <Key>>,
    class Allocator=allocator <pair  <const Key, Type>>>
class hash_multimap
```

Parameters

Key
The key data type to be stored in the hash_multimap.

Type
The element data type to be stored in the hash_multimap.

Traits
The type that includes two function objects, one of class *Traits* that is able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key>>` is the default value.

Allocator
The type that represents the stored allocator object that encapsulates details about the hash_multimap's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<pair <const Key, Type>>`.

Remarks

The hash_multimap is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Multiple, because its elements do not need to have a unique keys, so that one key value may have many element data values associated with it.
- A pair associative container, because its element values are distinct from its key values.
- A class template, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a hash_multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_multimap should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing,

say, definitions, where the words were not always uniquely defined. If, instead, the keywords were uniquely defined so that keys were unique, then a hash_map would be the container of choice. If, on the other hand, just the list of words were being stored, then a hash_set would be the correct container. If multiple occurrences of the words were allowed, then a hash_multiset would be the appropriate container structure.

The hash_multimap orders the sequence it controls by calling a stored hash `Traits` object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class hash_compare`<Key, less<Key>>`. Specifically, for all values `Key` of type `Key`, the call `Traits (Key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate f(x, y) is a function object that has two argument objects `x` and `y` and a return value of **`true`** or **`false`**. An ordering imposed on a hash_multimap is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects `x` and `y` are defined to be equivalent when both f(x, y) and f(y, x) are **`false`**. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_multimap class is a bidirectional iterator, but the class member functions insert and hash_multimap have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own hash_multimap of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal

hash_multimap of functionality, but it is enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the member functions.

Constructors

Constructor	Description
hash_multimap	Constructs a list of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other `hash_multimap`.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for the `hash_multimap` object.
const_iterator	A type that provides a bidirectional iterator that can read a **`const`** element in the `hash_multimap`.
const_pointer	A type that provides a pointer to a **`const`** element in a `hash_multimap`.
const_reference	A type that provides a reference to a **`const`** element stored in a `hash_multimap` for reading and performing **`const`** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **`const`** element in the `hash_multimap`.
difference_type	A signed integer type that can be used to represent the number of elements of a `hash_multimap` in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a `hash_multimap`.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `hash_multimap`.
key_type	A type that describes the sort key object that constitutes each element of the `hash_multimap`.
mapped_type	A type that represents the data type stored in a `hash_multimap`.
pointer	A type that provides a pointer to an element in a `hash_multimap`.
reference	A type that provides a reference to an element stored in a `hash_multimap`.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `hash_multimap`.

Type name	Description
size_type	An unsigned integer type that can represent the number of elements in a `hash_multimap`.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the `hash_multimap`.

Member functions

Member function	Description
begin	Returns an iterator addressing the first element in the `hash_multimap`.
cbegin	Returns a const iterator addressing the first element in the `hash_multimap`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `hash_multimap`.
clear	Erases all the elements of a `hash_multimap`.
count	Returns the number of elements in a `hash_multimap` whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed `hash_multimap`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `hash_multimap`.
emplace	Inserts an element constructed in place into a `hash_multimap`.
emplace_hint	Inserts an element constructed in place into a `hash_multimap`, with a placement hint.
empty	Tests if a `hash_multimap` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `hash_multimap`.
equal_range	Returns an iterator that addresses the location succeeding the last element in a `hash_multimap`.
erase	Removes an element or a range of elements in a `hash_multimap` from specified positions
find	Returns an iterator addressing the location of an element in a `hash_multimap` that has a key equivalent to a specified key.

Member function	Description
get_allocator	Returns a copy of the `allocator` object used to construct the `hash_multimap`.
insert	Inserts an element or a range of elements into the `hash_multimap` at a specified position.
key_comp	Retrieves a copy of the comparison object used to order keys in a `hash_multimap`.
lower_bound	Returns an iterator to the first element in a `hash_multimap` that with a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the `hash_multimap`.
rbegin	Returns an iterator addressing the first element in a reversed `hash_multimap`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `hash_multimap`.
size	Specifies a new size for a `hash_multimap`.
swap	Exchanges the elements of two `hash_multimap`s.
upper_bound	Returns an iterator to the first element in a `hash_multimap` that with a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a `hash_multimap`.

Operators

Operator	Description
hash_multimap::operator=	Replaces the elements of a `hash_multimap` with a copy of another `hash_multimap`.

Requirements

Header: <hash_map>

Namespace: stdext

hash_multimap::allocator_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that represents the allocator class for the hash_multimap object.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::allocator_type allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the hash_multimap Class topic.

Example

See the example for get_allocator for an example using `allocator_type`.

hash_multimap::begin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator addressing the first element in the hash_multimap.

```
C++
```

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_multimap or the location succeeding an empty hash_multimap.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_multimap object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_multimap object can be modified.

Example

C++

```
// hash_multimap_begin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 0, 0 ) );
   hm1.insert ( Int_Pair ( 1, 1 ) );
   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.begin ( );
   cout << "The first element of hm1 is " << hm1_cIter -> first
        << "." << endl;

   hm1_Iter = hm1.begin ( );
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.begin ( );
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.begin( );
   cout << "The first element of hm1 is now " << hm1_cIter -> first
        << "." << endl;
}
```

Output

```
The first element of hm1 is 0.
The first element of hm1 is now 1.
```

hash_multimap::cbegin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a const iterator addressing the first element in the hash_multimap.

C++

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the hash_multimap or the location succeeding an empty `hash_multimap`.

Example

C++

```
// hash_multimap_cbegin.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 2, 4 ) );

   hm1_cIter = hm1.cbegin ( );
   cout << "The first element of hm1 is "
        << hm1_cIter -> first << "." << endl;
   }
```

Output

```
The first element of hm1 is 2.
```

hash_multimap::cend

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a const iterator that addresses the location succeeding the last element in a hash_multimap.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a hash_multimap. If the `hash_multimap` is empty, then `hash_multimap::cend == hash_multimap::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its hash_multimap.

The value returned by `cend` should not be dereferenced.

Example

```
C++
```

```
// hash_multimap_cend.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: const_iterator hm1_cIter;
```

```
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.cend( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;
   }
```

Output

```
The value of last element of hm1 is 30.
```

hash_multimap::clear

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Erases all the elements of a hash_multimap.

C++

```
void clear();
```

Remarks

Example

The following example demonstrates the use of the hash_multimap::clear member function.

C++

```
// hash_multimap_clear.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
```

```
    hash_multimap<int, int> hm1;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 4));

    i = hm1.size();
    cout << "The size of the hash_multimap is initially "
         << i  << "." << endl;

    hm1.clear();
    i = hm1.size();
    cout << "The size of the hash_multimap after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the hash_multimap is initially 2.
The size of the hash_multimap after clearing is 0.
```

hash_multimap::const_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a bidirectional iterator that can read a **`const`** element in the hash_multimap.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

The `const_iterator` defined by hash_multimap points to objects of value_type, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair, and the value of the mapped element is available through the second member of the pair.

To dereference a `const_iterator` `cIter` pointing to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See the example for begin for an example using `const_iterator`.

hash_multimap::const_pointer

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a pointer to a **`const`** element in a hash_multimap.

```
C++

typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_multimap object.

hash_multimap::const_reference

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a reference to a **`const`** element stored in a hash_multimap for reading and performing **`const`** operations.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_reference const_reference;
```

Remarks

Example

C++

```
// hash_multimap_const_ref.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap<int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin() -> second );

   cout << "The data value of 1st element in the hash_multimap is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of 1st element in the hash_multimap is 10.
```

hash_multimap::const_reverse_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a bidirectional iterator that can read any **`const`** element in the hash_multimap.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_reverse_iterator const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_multimap in reverse.

The `const_reverse_iterator` defined by hash_multimap points to objects of value_type, which are of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator` `crIter` pointing to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `crIter->first`, which is equivalent to `(*crIter).first`. To access the value of the mapped datum for the element, use `crIter->second`, which is equivalent to `(*crIter).second`.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

hash_multimap::count

> ⓘ Note

This API is obsolete. The alternative is **unordered_multimap Class**.

Returns the number of elements in a hash_multimap whose key matches a parameter-specified key.

C++

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the hash_multimap.

Return Value

1 if the hash_multimap contains an element whose sort key matches the parameter key; 0 if the hash_multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

[lower_bound (`key`), upper_bound (`key`))

which have a key value *key*.

Example

The following example demonstrates the use of the hash_multimap::count member function.

C++

```
// hash_multimap_count.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
```

```
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    hm1.insert(Int_Pair(2, 1));
    hm1.insert(Int_Pair(1, 4));
    hm1.insert(Int_Pair(2, 1));

    // Elements do not need to have unique keys in hash_multimap,
    // so duplicates are allowed and counted
    i = hm1.count(1);
    cout << "The number of elements in hm1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hm1.count(2);
    cout << "The number of elements in hm1 with a sort key of 2 is: "
         << i << "." << endl;

    i = hm1.count(3);
    cout << "The number of elements in hm1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hm1 with a sort key of 1 is: 2.
The number of elements in hm1 with a sort key of 2 is: 2.
The number of elements in hm1 with a sort key of 3 is: 0.
```

hash_multimap::crbegin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a const iterator addressing the first element in a reversed hash_multimap.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed hash_multimap or addressing what had been the last element in the unreversed

`hash_multimap`.

Remarks

`crbegin` is used with a reversed hash_multimap just as hash_multimap::begin is used with a `hash_multimap`.

With the return value of `crbegin`, the `hash_multimap` object cannot be modified.

`crbegin` can be used to iterate through a `hash_multimap` backwards.

Example

C++

```
// hash_multimap_crbegin.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_crIter = hm1.crbegin( );
   cout << "The first element of the reversed hash_multimap hm1 is "
        << hm1_crIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_multimap hm1 is 3.
```

hash_multimap::crend

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_multimap.

```
C++
```

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multimap (the location that had preceded the first element in the unreversed `hash_multimap`).

Remarks

`crend` is used with a reversed hash_multimap just as hash_multimap::end is used with a hash_multimap.

With the return value of `crend`, the `hash_multimap` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its hash_multimap.

The value returned by `crend` should not be dereferenced.

Example

```
C++
```

```
// hash_multimap_crend.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 3, 30 ) );
```

```
   hm1_crIter = hm1.crend( );
   hm1_crIter--;
   cout << "The last element of the reversed hash_multimap hm1 is "
        << hm1_crIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_multimap hm1 is 3.
```

hash_multimap::difference_type

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A signed integer type that can be used to represent the number of elements of a hash_multimap in a range between elements pointed to by iterators.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range *[first, last)* between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

C++

```
// hash_multimap_difference_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>
#include <algorithm>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(2, 20));
    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(3, 20));

    // The following will insert, because map keys
    // do not need to be unique
    hm1.insert(Int_Pair(2, 30));

    hash_multimap<int, int>::iterator hm1_Iter, hm1_bIter, hm1_eIter;
    hm1_bIter = hm1.begin();
    hm1_eIter = hm1.end();

    // Count the number of elements in a hash_multimap
    hash_multimap<int, int>::difference_type df_count = 0;
    hm1_Iter = hm1.begin();
    while (hm1_Iter != hm1_eIter)
    {
        df_count++;
        hm1_Iter++;
    }

    cout << "The number of elements in the hash_multimap hm1 is: "
         << df_count << "." << endl;

    cout << "The keys of the mapped elements are:";
    for (hm1_Iter= hm1.begin() ; hm1_Iter!= hm1.end();
        hm1_Iter++)
        cout << " " << hm1_Iter-> first;
    cout << "." << endl;

    cout << "The values of the mapped elements are:";
    for (hm1_Iter= hm1.begin() ; hm1_Iter!= hm1.end();
        hm1_Iter++)
        cout << " " << hm1_Iter-> second;
    cout << "." << endl;
}
```

Output

```
The number of elements in the hash_multimap hm1 is: 4.
The keys of the mapped elements are: 1 2 2 3.
The values of the mapped elements are: 10 20 30 20.
```

hash_multimap::emplace

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Inserts an element constructed in place into a hash_multimap.

C++

```
template <class ValTy>
iterator emplace(ValTy&& val);
```

Parameters

val
The value used to move construct an element to be inserted into the hash_multimap.

Return Value

The `emplace` member function returns an iterator that points to the position where the new element was inserted.

Remarks

The hash_multimap::value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

C++

```
// hash_multimap_emplace.cpp
// compile with: /EHsc
#include<hash_multimap>
```

```
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(move(is1));
    cout << "After the emplace, hm1 contains:" << endl
      << " " << hm1.begin()->first
      << " => " << hm1.begin()->second
      << endl;
}
```

Output

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_multimap::emplace_hint

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Inserts an element constructed in place into a hash_multimap, with a placement hint.

C++

```
template <class ValTy>
iterator emplace_hint(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

val
The value used to move construct an element to be inserted into the hash_multimap unless the `hash_multimap` already contains that element (or, more generally, an element whose key is equivalently ordered).

_Where
A hint regarding the place to start searching for the correct point of insertion.

Return Value

The hash_multimap::emplace member function returns an iterator that points to the position where the new element was inserted into the `hash_multimap`.

Remarks

The hash_multimap::value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

C++

```
// hash_multimap_emplace_hint.cpp
// compile with: /EHsc
#include<hash_multimap>
#include<iostream>
#include <string>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, string> hm1;
    typedef pair<int, string> is1(1, "a");

    hm1.emplace(hm1.begin(), move(is1));
    cout << "After the emplace insertion, hm1 contains:" << endl
      << " " << hm1.begin()->first
      << " => " << hm1.begin()->second
      << endl;
}
```

Output

```
After the emplace insertion, hm1 contains:
1 => a
```

hash_multimap::empty

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Tests if a hash_multimap is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the hash_multimap is empty; `false` if the hash_multimap is nonempty.

Remarks

Example

C++

```
// hash_multimap_empty.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2;

   typedef pair <int, int> Int_Pair;
   hm1.insert ( Int_Pair ( 1, 1 ) );

   if ( hm1.empty( ) )
      cout << "The hash_multimap hm1 is empty." << endl;
   else
      cout << "The hash_multimap hm1 is not empty." << endl;

   if ( hm2.empty( ) )
      cout << "The hash_multimap hm2 is empty." << endl;
   else
```

```
        cout << "The hash_multimap hm2 is not empty." << endl;
}
```

Output

```
The hash_multimap hm1 is not empty.
The hash_multimap hm2 is empty.
```

hash_multimap::end

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator that addresses the location succeeding the last element in a hash_multimap.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multimap. If the hash_multimap is empty, then hash_multimap::end == hash_multimap::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_multimap.

The value returned by `end` should not be dereferenced.

Example

C++

```
// hash_multimap_end.cpp
// compile with: /EHsc
```

```
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: const_iterator hm1_cIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is "
        << hm1_cIter -> second << "." << endl;

   hm1_Iter = hm1.end( );
   hm1_Iter--;
   hm1.erase ( hm1_Iter );

   // The following 2 lines would err because the iterator is const
   // hm1_cIter = hm1.end ( );
   // hm1_cIter--;
   // hm1.erase ( hm1_cIter );

   hm1_cIter = hm1.end( );
   hm1_cIter--;
   cout << "The value of last element of hm1 is now "
        << hm1_cIter -> second << "." << endl;
}
```

Output

```
The value of last element of hm1 is 30.
The value of last element of hm1 is now 20.
```

hash_multimap::equal_range

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a pair of iterators respectively to the first element in a hash_multimap with a key that is greater than a specified key and to the first element in the hash_multimap with a key that is equal to or greater than the key.

```
C++
```

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **first** and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second** and to dereference the upper bound iterator, use *(`pr`. **second**).

Remarks

Example

```
C++
```

```
// hash_multimap_equal_range.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_multimap <int, int> IntMMap;
   IntMMap hm1;
   hash_multimap <int, int> :: const_iterator hm1_RcIter;
```

```
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
   p1 = hm1.equal_range( 2 );

   cout << "The lower bound of the element with a key of 2\n"
        << "in the hash_multimap hm1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with a key of 2\n"
        << "in the hash_multimap hm1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   hm1_RcIter = hm1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << hm1_RcIter -> second << "," << endl
        << "matching the 2nd element of the pair "
        << "returned by equal_range( 2 )." << endl;

   p2 = hm1.equal_range( 4 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == hm1.end( ) ) && ( p2.second == hm1.end( ) ) )
      cout << "The hash_multimap hm1 doesn't have an element "
           << "with a key less than 4." << endl;
   else
      cout << "The element of hash_multimap hm1 with a key >= 40 is: "
           << p1.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2
in the hash_multimap hm1 is: 20.
The upper bound of the element with a key of 2
in the hash_multimap hm1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 2 ).
The hash_multimap hm1 doesn't have an element with a key less than 4.
```

hash_multimap::erase

ⓘ Note

This API is obsolete. The alternative is **unordered_multimap Class**.

Removes an element or a range of elements in a hash_multimap from specified positions or removes elements that match a specified key.

```
C++

iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

_Where
Position of the element to be removed from the hash_multimap.

first
Position of the first element removed from the hash_multimap.

last
Position just beyond the last element removed from the hash_multimap.

key
The key of the elements to be removed from the hash_multimap.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_multimap if no such element exists.

For the third member function, returns the number of elements that have been removed from the hash_multimap.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the hash_multimap::erase member function.

C++

```
// hash_multimap_erase.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2, hm3;
    hash_multimap<int, int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_multimap<int, int>::size_type n;
    typedef pair<int, int> Int_Pair;

    for (i = 1; i < 5; i++)
    {
        hm1.insert(Int_Pair (i, i) );
        hm2.insert(Int_Pair (i, i*i) );
        hm3.insert(Int_Pair (i, i-1) );
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hm1.begin();
    hm1.erase(Iter1);

    cout << "After the 2nd element is deleted, "
         << "the hash_multimap hm1 is:";
    for (pIter = hm1.begin(); pIter != hm1.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [ first,  last)
    Iter1 = ++hm2.begin();
    Iter2 = --hm2.end();
    hm2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_multimap hm2 is:";
    for (pIter = hm2.begin(); pIter != hm2.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function removes elements with a given  key
    hm3.insert(Int_Pair (2, 5));
    n = hm3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
```

```
         << "the hash_multimap hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hm3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hm3.begin();
    hm3.erase(Iter1);

    cout << "After another element with a key equal to that of the"
         << endl;
    cout  << "2nd element is deleted, "
          << "the hash_multimap hm3 is:";
    for (pIter = hm3.begin(); pIter != hm3.end(); pIter++)
        cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_multimap hm1 is: 1 3 4.
After the middle two elements are deleted, the hash_multimap hm2 is: 1 16.
After the element with a key of 2 is deleted,
the hash_multimap hm3 is: 0 2 3.
The number of elements removed from hm3 is: 2.
After another element with a key equal to that of the
2nd element is deleted, the hash_multimap hm3 is: 0 3.
```

hash_multimap::find

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator addressing the first location of an element in a hash_multimap that has a key equivalent to a specified key.

C++

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key
The key to be matched by the sort key of an element from the hash_multimap being searched.

Return Value

An iterator that addresses the first location of an element with a specified key, or the location succeeding the last element in the hash_multimap if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multimap whose sort key is `equivalent` to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `find` is assigned to an `iterator`, the hash_multimap object can be modified.

Example

C++

```
// hash_multimap_find.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_map>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1;
    hash_multimap<int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 10));
    hm1.insert(Int_Pair(2, 20));
    hm1.insert(Int_Pair(3, 20));
    hm1.insert(Int_Pair(3, 30));

    hm1_RcIter = hm1.find(2);
    cout << "The element of hash_multimap hm1 with a key of 2 is: "
```

```
          << hm1_RcIter -> second << "." << endl;

    hm1_RcIter = hm1.find(3);
    cout << "The first element of hash_multimap hm1 with a key of 3 is: "
          << hm1_RcIter -> second << "." << endl;

    // If no match is found for the key, end() is returned
    hm1_RcIter = hm1.find(4);

    if (hm1_RcIter == hm1.end())
        cout << "The hash_multimap hm1 doesn't have an element "
             << "with a key of 4." << endl;
    else
        cout << "The element of hash_multimap hm1 with a key of 4 is: "
             << hm1_RcIter -> second << "." << endl;

    // The element at a specific location in the hash_multimap can be
    // found using a dereferenced iterator addressing the location
    hm1_AcIter = hm1.end();
    hm1_AcIter--;
    hm1_RcIter = hm1.find(hm1_AcIter -> first);
    cout << "The first element of hm1 with a key matching"
         << endl << "that of the last element is: "
         << hm1_RcIter -> second << "." << endl;

    // Note that the first element with a key equal to
    // the key of the last element is not the last element
    if (hm1_RcIter == --hm1.end())
        cout << "This is the last element of hash_multimap hm1."
             << endl;
    else
        cout << "This is not the last element of hash_multimap hm1."
             << endl;
}
```

Output

```
The element of hash_multimap hm1 with a key of 2 is: 20.
The first element of hash_multimap hm1 with a key of 3 is: 20.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key matching
that of the last element is: 20.
This is not the last element of hash_multimap hm1.
```

hash_multimap::get_allocator

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns a copy of the allocator object used to construct the hash_multimap.

```
C++
```

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_multimap.

Remarks

Allocators for the hash_multimap class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
C++
```

```
// hash_multimap_get_allocator.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int>::allocator_type hm1_Alloc;
   hash_multimap <int, int>::allocator_type hm2_Alloc;
   hash_multimap <int, double>::allocator_type hm3_Alloc;
   hash_multimap <int, int>::allocator_type hm4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> hm2;
   hash_multimap <int, double> hm3;

   hm1_Alloc = hm1.get_allocator( );
   hm2_Alloc = hm2.get_allocator( );
   hm3_Alloc = hm3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << " before free memory is exhausted: "
```

```
        << hm2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << " before free memory is exhausted: "
        << hm3.max_size( ) <<  "." << endl;

   // The following line creates a hash_multimap hm4
   // with the allocator of hash_multimap hm1.
   hash_multimap <int, int> hm4( less<int>( ), hm1_Alloc );

   hm4_Alloc = hm4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hm1_Alloc == hm4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

hash_multimap::hash_multimap

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Constructs a hash_multimap that is empty or is a copy of all or part of some other hash_multimap.

C++

```
hash_multimap();

explicit hash_multimap(
    const Compare& Comp);

hash_multimap(
    const Compare& Comp,
    const Allocator& Al);

hash_multimap(
    const hash_multimap& Right);
```

```
hash_multimap(
    hash_multimap&& Right);

hash_multimap(
    initializer_list<Type> IList);

hash_multimap(
    initializer_list<Type> IList,
    const Compare& Comp);

hash_multimap(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last,
    const Compare& Comp);

template <class InputIterator>
hash_multimap(
    InputIterator First,
    InputIterator Last,
    const Compare& Comp,
    const Allocator& Al);
```

Parameters

Al

The storage allocator class to be used for this hash_multimap object, which defaults to `Allocator`.

Comp

The comparison function of type `const Traits` used to order the elements in the map, which defaults to `Traits`.

Right

The map of which the constructed set is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last
The position of the first element beyond the range of elements to be copied.

IList
The initializer_list to copy from.

Remarks

All constructors store a type of allocator object that manages memory storage for the hash_multimap and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros are used to substitute alternative allocators.

All constructors initialize their hash_multimap.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the hash_multimap and can later be returned by calling key_comp.

The first three constructors specify an empty initial hash_multimap; the second specifies the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifies the allocator type (`_Al`) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the hash_multimap `Right`.

The next three constructors copy the range `First, Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

The eighth constructor moves the hash_multimap `Right`.

The final three constructors use an initializer_list.

hash_multimap::insert

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Inserts an element or a range of elements into a hash_multimap.

```
C++
```

```
iterator insert(
    const value_type& Val);

iterator insert(
    const_iterator Where,
    const value_type& Val);void insert(
    initializer_list<value_type> IList);

template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

template <class ValTy>
iterator insert(
    ValTy&& Val);

template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);
```

Parameters

Val
The value of an element to be inserted into the hash_multimap unless it already contains that element, or more generally, unless it already contains an element whose key is equivalently ordered.

Where
A hint about where to start searching for the correct point of insertion.

First
The position of the first element to be copied from a map.

Last
The position just beyond the last element to be copied from a map.

Return Value

The first two `insert` member functions return an iterator that points to the position where the new element was inserted.

The third member function uses an initializer_list for the elements to be inserted.

The fourth member function inserts the sequence of element values into a map that corresponds to each element addressed by an iterator in the range `[First, Last)` of a specified set.

The last two `insert` member functions behave the same as the first two, except that they move-construct the inserted value.

Remarks

The value_type of an element is a pair, so that the value of an element will be an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

Insertion can occur in amortized constant time for the hint version of `insert`, instead of logarithmic time, if the insertion point immediately follows *Where*.

hash_multimap::iterator

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a bidirectional iterator that can read or modify any element in a hash_multimap.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::iterator iterator;
```

Remarks

The `iterator` defined by hash_multimap points to objects of value_type, which are of type `pair`< **const Key, Type**>, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an iterator named `Iter` that points to an element in a hash_multimap, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).first`.

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin for an example of how to declare and use `iterator`.

hash_multimap::key_comp

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Retrieves a copy of the comparison object used to order keys in a hash_multimap.

```
C++
```

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a hash_multimap uses to order its elements.

Remarks

The stored object defines the member function

`bool operator( const Key& left, const Key& right );`

which returns **`true`** if `left` precedes and is not equal to `right` in the sort order.

Example

```
C++
```

```
// hash_multimap_key_comp.cpp
// compile with: /EHsc
#include <hash_map>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multimap <int, int, hash_compare<int, less<int>>> hm1;
   hash_multimap <int, int, hash_compare<int, less<int>>
      >::key_compare kc1 = hm1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true,\n"
           << "where kc1 is the function object of hm1.\n"
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false,\n"
           << "where kc1 is the function object of hm1.\n"
           << endl;
   }

   hash_multimap <int, int, hash_compare<int, greater<int>>> hm2;
   hash_multimap <int, int, hash_compare<int, greater<int>>
      >::key_compare kc2 = hm2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true,\n"
           << "where kc2 is the function object of hm2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false,\n"
           << "where kc2 is the function object of hm2."
           << endl;
   }
}
```

hash_multimap::key_compare

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_multimap.

```
C++

typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the hash_multimap Class topic.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

hash_multimap::key_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that describes the sort key object that constitutes each element of the hash_multimap.

```
C++

typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

For more information on *Key*, see the Remarks section of the hash_multimap Class topic.

Example

See the example for value_type for an example of how to declare and use `key_compare`.

hash_multimap::lower_bound

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator to the first element in a hash_multimap with a key that is equal to or greater than a specified key.

```
C++
```

```
iterator lower_bound(const Key& key);

const_iterator lower_bound(const Key& key) const;
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_multimap with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `lower_bound` is assigned to an `iterator`, the hash_multimap object can be modified.

Remarks

Example

```
C++
```

```
// hash_multimap_lower_bound.cpp
// compile with: /EHsc
#include <hash_map>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: const_iterator hm1_AcIter,
      hm1_RcIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_RcIter = hm1.lower_bound( 2 );
   cout << "The element of hash_multimap hm1 with a key of 2 is: "
        << hm1_RcIter -> second << "." << endl;

   hm1_RcIter = hm1.lower_bound( 3 );
   cout << "The first element of hash_multimap hm1 with a key of 3 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.lower_bound( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_multimap hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_multimap hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_multimap can be
   // found using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.end( );
   hm1_AcIter--;
   hm1_RcIter = hm1.lower_bound( hm1_AcIter -> first );
   cout << "The first element of hm1 with a key matching"
        << endl << "that of the last element is: "
        << hm1_RcIter -> second << "." << endl;

   // Note that the first element with a key equal to
   // the key of the last element is not the last element
   if ( hm1_RcIter == --hm1.end( ) )
      cout << "This is the last element of hash_multimap hm1."
           << endl;
   else
      cout << "This is not the last element of hash_multimap hm1."
           << endl;
}
```

Output

```
The element of hash_multimap hm1 with a key of 2 is: 20.
The first element of hash_multimap hm1 with a key of 3 is: 20.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key matching
that of the last element is: 20.
This is not the last element of hash_multimap hm1.
```

hash_multimap::mapped_type

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that represents the data type stored in a hash_multimap.

C++

```
typedef Type mapped_type;
```

Remarks

`mapped_type` is a synonym for the template parameter *Type*.

For more information on *Type* see the hash_multimap Class topic.

Example

See the example for value_type for an example of how to declare and use `key_type`.

hash_multimap::max_size

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns the maximum length of the hash_multimap.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_multimap.

Remarks

Example

C++

```
// hash_multimap_max_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: size_type i;

   i = hm1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_multimap is " << i << "." << endl;
}
```

hash_multimap::operator=

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Replaces the elements of the hash_multimap with a copy of another hash_multimap.

C++

```
hash_multimap& operator=(const hash_multimap& right);

hash_multimap& operator=(hash_multimap&& right);
```

Parameters

right
The hash_multimap being copied into the `hash_multimap`.

Remarks

After erasing any existing elements in a `hash_multimap`, `operator=` either copies or moves the contents of *right* into the `hash_multimap`.

Example

C++

```
// hash_multimap_operator_as.cpp
// compile with: /EHsc
#include <hash_multimap>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap<int, int> v1, v2, v3;
   hash_multimap<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
}
```

hash_multimap::pointer

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a pointer to an element in a hash_multimap.

```
C++
```

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_multimap object.

hash_multimap::rbegin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator addressing the first element in a reversed hash_multimap.

```
C++
```

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multimap or addressing what had been the last element in the unreversed hash_multimap.

Remarks

`rbegin` is used with a reversed hash_multimap just as begin is used with a hash_multimap.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_multimap object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_multimap object can be modified.

`rbegin` can be used to iterate through a hash_multimap backwards.

Example

C++

```
// hash_multimap_rbegin.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: reverse_iterator hm1_rIter;
   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rbegin( );
   cout << "The first element of the reversed hash_multimap hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_multimap in a forward order
   cout << "The hash_multimap is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_multimap in a reverse order
   cout << "The reversed hash_multimap is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
```

```
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;

   // A hash_multimap element can be erased by dereferencing its key
   hm1_rIter = hm1.rbegin( );
   hm1.erase ( hm1_rIter -> first );

   hm1_rIter = hm1.rbegin( );
   cout << "After the erasure, the first element\n"
        << "in the reversed hash_multimap is "
        << hm1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed hash_multimap hm1 is 3.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the first element
in the reversed hash_multimap is 2.
```

hash_multimap::reference

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that provides a reference to an element stored in a hash_multimap.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::reference reference;
```

Remarks

Example

C++

```
// hash_multimap_reference.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( hm1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference cannot be used to access the key
   // int &Ref1 = ( hm1.begin( ) -> first );

   cout << "The key of first element in the hash_multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( hm1.begin( ) -> second );

   cout << "The data value of first element in the hash_multimap is "
        << Ref2 << "." << endl;

   //The non-const_reference can be used to modify the
   //data value of the first element
   Ref2 = Ref2 + 5;
   cout << "The modified data value of first element is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The modified data value of first element is 15.
```

hash_multimap::rend

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multimap.

```
C++
```

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multimap (the location that had preceded the first element in the unreversed hash_multimap).

Remarks

`rend` is used with a reversed hash_multimap just as end is used with a hash_multimap.

If the return value of `rend` is assigned to a const_reverse_iterator, then the hash_multimap object cannot be modified. If the return value of `rend` is assigned to a reverse_iterator, then the hash_multimap object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its hash_multimap.

The value returned by `rend` should not be dereferenced.

Example

```
C++
```

```
// hash_multimap_rend.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;

   hash_multimap <int, int> :: iterator hm1_Iter;
   hash_multimap <int, int> :: reverse_iterator hm1_rIter;
```

```
   hash_multimap <int, int> :: const_reverse_iterator hm1_crIter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "The last element of the reversed hash_multimap hm1 is "
        << hm1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a hash_multimap in a forward order
   cout << "The hash_multimap is: ";
   for ( hm1_Iter = hm1.begin( ) ; hm1_Iter != hm1.end( ); hm1_Iter++)
      cout << hm1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a hash_multimap in a reverse order
   cout << "The reversed hash_multimap is: ";
   for ( hm1_rIter = hm1.rbegin( ) ; hm1_rIter != hm1.rend( ); hm1_rIter++)
      cout << hm1_rIter -> first << " ";
      cout << "." << endl;

   // A hash_multimap element can be erased by dereferencing its key
   hm1_rIter = --hm1.rend( );
   hm1.erase ( hm1_rIter -> first );

   hm1_rIter = hm1.rend( );
   hm1_rIter--;
   cout << "After the erasure, the last element "
        << "in the reversed hash_multimap is "
        << hm1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed hash_multimap hm1 is 1.
The hash_multimap is: 1 2 3 .
The reversed hash_multimap is: 3 2 1 .
After the erasure, the last element in the reversed hash_multimap is 2.
```

hash_multimap::reverse_iterator

> ⓘ **Note**
>
> This API is obsolete. The alternative is unordered_multimap Class.

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multimap.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::reverse_iterator reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the hash_multimap in reverse.

The `reverse_iterator` defined by hash_multimap points to objects of value_type, which are of type `pair` < **const Key, Type**>. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

Example

See the example for rbegin for an example of how to declare and use `reverse_iterator`.

hash_multimap::size

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Returns the number of elements in the hash_multimap.

```
C++
```

```
size_type size() const;
```

Return Value

The current length of the hash_multimap.

Remarks

Example

The following example demonstrates the use of the hash_multimap::size member function.

C++

```
// hash_multimap_size.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multimap<int, int> hm1, hm2;
    hash_multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    hm1.insert(Int_Pair(1, 1));
    i = hm1.size();
    cout << "The hash_multimap length is " << i << "." << endl;

    hm1.insert(Int_Pair(2, 4));
    i = hm1.size();
    cout << "The hash_multimap length is now " << i << "." << endl;
}
```

Output

```
The hash_multimap length is 1.
The hash_multimap length is now 2.
```

hash_multimap::size_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

An unsigned integer type that counts the number of elements in a hash_multimap.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See the example for size for an example of how to declare and use `size_type`

hash_multimap::swap

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

Exchanges the elements of two hash_multimaps.

```
C++
```

```
void swap(hash_multimap& right);
```

Parameters

right
The hash_multimap providing the elements to be swapped or the hash_multimap whose elements are to be exchanged with those of the hash_multimap.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multimaps whose elements are being exchanged.

Example

```
C++
```

```
// hash_multimap_swap.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
```

```
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1, hm2, hm3;
   hash_multimap <int, int>::iterator hm1_Iter;
   typedef pair <int, int> Int_Pair;

   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm2.insert ( Int_Pair ( 10, 100 ) );
   hm2.insert ( Int_Pair ( 20, 200 ) );
   hm3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   hm1.swap( hm2 );

   cout << "After swapping with hm2, hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hm1, hm3 );

   cout << "After swapping with hm3, hash_multimap hm1 is:";
   for ( hm1_Iter = hm1.begin( ); hm1_Iter != hm1.end( ); hm1_Iter++ )
      cout << " " << hm1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original hash_multimap hm1 is: 10 20 30.
After swapping with hm2, hash_multimap hm1 is: 100 200.
After swapping with hm3, hash_multimap hm1 is: 300.
```

hash_multimap::upper_bound

Note

This API is obsolete. The alternative is **unordered_multimap Class**.

Returns an iterator to the first element in a hash_multimap with a key that is greater than a specified key.

```
C++
```

```
iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multimap being searched.

Return Value

An iterator or const_iterator that addresses the location of an element in a hash_multimap with a key that is greater than the argument key, or that addresses the location succeeding the last element in the hash_multimap if no match is found for the key.

If the return value of `upper_bound` is assigned to a `const_iterator`, the hash_multimap object cannot be modified. If the return value of `upper_bound` is assigned to a `iterator`, the hash_multimap object can be modified.

Remarks

Example

```
C++
```

```
// hash_multimap_upper_bound.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: const_iterator hm1_AcIter, hm1_RcIter;
   typedef pair <int, int> Int_Pair;
```

```
   hm1.insert ( Int_Pair ( 1, 10 ) );
   hm1.insert ( Int_Pair ( 2, 20 ) );
   hm1.insert ( Int_Pair ( 3, 30 ) );
   hm1.insert ( Int_Pair ( 3, 40 ) );

   hm1_RcIter = hm1.upper_bound( 1 );
   cout << "The 1st element of hash_multimap hm1 with "
        << "a key greater than 1 is: "
        << hm1_RcIter -> second << "." << endl;

   hm1_RcIter = hm1.upper_bound( 2 );
   cout << "The first element of hash_multimap hm1\n"
        << "with a key greater than 2 is: "
        << hm1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   hm1_RcIter = hm1.lower_bound( 4 );

   if ( hm1_RcIter == hm1.end( ) )
      cout << "The hash_multimap hm1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of hash_multimap hm1 with a key of 4 is: "
           << hm1_RcIter -> second << "." << endl;

   // The element at a specific location in the hash_multimap can be
   // found using a dereferenced iterator addressing the location
   hm1_AcIter = hm1.begin( );
   hm1_RcIter = hm1.upper_bound( hm1_AcIter -> first );
   cout << "The first element of hm1 with a key greater than"
        << endl << "that of the initial element of hm1 is: "
        << hm1_RcIter -> second << "." << endl;
}
```

Output

```
The 1st element of hash_multimap hm1 with a key greater than 1 is: 20.
The first element of hash_multimap hm1
with a key  greater than 2 is: 30.
The hash_multimap hm1 doesn't have an element with a key of 4.
The first element of hm1 with a key greater than
that of the initial element of hm1 is: 20.
```

hash_multimap::value_comp

Note

This API is obsolete. The alternative is unordered_multimap Class.

The member function returns a function object that determines the order of elements in a hash_multimap by comparing their key values.

```
C++
```

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a hash_multimap uses to order its elements.

Remarks

For a hash_multimap *m*, if two elements *e1* (*k1*, *d1*) and *e2*(*k2*, *d2*) are objects of type value_type, where *k1* and *k2* are their keys of type key_type and *d1* and *d2* are their data of type mapped_type, then `m.value_comp()(e1, e2)` is equivalent to `m.key_comp()(k1, k2)`. A stored object defines the member function

`bool operator( value_type& left, value_type& right);`

which returns **`true`** if the key value of `left` precedes and is not equal to the key value of `right` in the sort order.

Example

```
C++
```

```
// hash_multimap_value_comp.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multimap <int, int, hash_compare<int, less<int>>> hm1;
   hash_multimap <int, int, hash_compare<int, less<int>>
      >::value_compare vc1 = hm1.value_comp( );
   hash_multimap <int,int>::iterator Iter1, Iter2;

   Iter1= hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );
   Iter2= hm1.insert ( hash_multimap <int, int> :: value_type ( 2, 5 ) );
```

```
   if( vc1( *Iter1, *Iter2 ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does "
           << "not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1( *Iter2, *Iter1 ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does "
           << "not precede the element ( 1,10 )."
           << endl;
   }
}
```

hash_multimap::value_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_multimap Class**.

A type that represents the type of object stored in a hash_multimap.

C++

```
typedef pair<const Key, Type> value_type;
```

Remarks

`value_type` is declared to be pair<const key_type, mapped_type> and not pair<key_type, mapped_type> because the keys of an associative container may not be changed using a nonconstant iterator or reference.

Example

C++

```cpp
// hash_multimap_value_type.cpp
// compile with: /EHsc
#include <hash_map>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef pair <const int, int> cInt2Int;
   hash_multimap <int, int> hm1;
   hash_multimap <int, int> :: key_type key1;
   hash_multimap <int, int> :: mapped_type mapped1;
   hash_multimap <int, int> :: value_type value1;
   hash_multimap <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitly to avoid implicit type conversion
   hm1.insert ( hash_multimap <int, int> :: value_type ( 1, 10 ) );

   // Compare another way to insert objects into a hash_multimap
   hm1.insert ( cInt2Int ( 2, 20 ) );

   // Initializing key1 and mapped1
   key1 = ( hm1.begin( ) -> first );
   mapped1 = ( hm1.begin( ) -> second );

   cout << "The key of first element in the hash_multimap is "
        << key1 << "." << endl;

   cout << "The data value of first element in the hash_multimap is "
        << mapped1 << "." << endl;

   // The following line would cause an error because
   // the value_type is not assignable
   // value1 = cInt2Int ( 4, 40 );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = hm1.begin( ) ; pIter != hm1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;
}
```

Output

```
The key of first element in the hash_multimap is 1.
The data value of first element in the hash_multimap is 10.
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

value_compare Class

Article • 08/03/2021

Provides a function object that can compare the elements of a hash_map by comparing the values of their keys to determine their relative order in the hash_map.

Syntax

C++

```
class value_compare
    : std::public binary_function<value_type, value_type, bool>
{
public:
    bool operator()(
        const value_type& left,
        const value_type& right) const
    {
        return (comp(left.first, right.first));
    }

protected:
    value_compare(const key_compare& c) : comp (c) { }
    key_compare comp;
};
```

Remarks

The comparison criteria provided by value_compare between `value_types` of whole elements contained by a hash_map is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object `comp` of type `key_compare` stored in the function object provided by value_compare to compare the sort-key components of two elements.

For hash_sets and hash_multisets, which are simple containers where the key values are identical to the element values, value_compare is equivalent to `key_compare`; for hash_maps and hash_multimaps they are not, because the value of the type `pair` elements is not identical to the value of the element's key.

Example

See the example for hash_map::value_comp for an example of how to declare and use value_compare.

Requirements

Header: <hash_map>

Namespace: stdext

See also

binary_function Struct
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

<hash_set>

Article • 12/07/2021

> **Note**
>
> This header is obsolete. The alternative is <unordered_set>.

Defines the container class templates hash_set and hash_multiset and their supporting templates.

Syntax

```
C++

#include <hash_set>
```

Remarks

Operators

Hash_set version	Hash_multiset version	Description
operator!= (hash_set)	operator!= (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is not equal to the hash_set or hash_multiset object on the right side.
operator== (hash_set)	operator== (hash_multiset)	Tests if the hash_set or hash_multiset object on the left side of the operator is equal to the hash_set or hash_multiset object on the right side.

Specialized Template Functions

Hash_set version	Hash_multiset version	Description
swap (hash_set)	swap (hash_multiset)	Exchanges the elements of two hash_sets or hash_multisets.

Classes

Class	Description
hash_compare Class	Describes an object that can be used by any of the hash associative containers — hash_map, hash_multimap, hash_set, or hash_multiset — as a default `Traits` parameter object to order and hash the elements they contain.
hash_set Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.
hash_multiset Class	Used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

hash_set Class

Article • 08/03/2021

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

The container class hash_set is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values.

Syntax

C++

```
template <class Key,
    class Traits=hash_compare<Key, less<Key>>,
    class Allocator=allocator<Key>>
class hash_set
```

Parameters

Key
The element data type to be stored in the hash_set.

Traits
The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.

Allocator
The type that represents the stored allocator object that encapsulates details about the hash_set's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

The hash_set is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because hash_set is also a simple associative container, its elements are also unique.
- A class template because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency; a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a hash_set are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a hash_multiset would be the appropriate container structure. If values need to be attached to a list of unique key words, then a

hash_map would be an appropriate structure to contain this data. If instead the keys are not unique, then a hash_multimap would be the container of choice.

The hash_set orders the sequence it controls by calling a stored hash `Traits` object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class *hash_compare<Key, less<Key> >*. Specifically, for all values `key` of type Key, the call Trait(`key`) yields a distribution of values of type size_t.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the non-equivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*, *y*) is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a hash_set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f*(*x*, *y*) and *f*(*y*, *x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_set class is a bidirectional iterator, but the class member functions insert and hash_set have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`first`, `last`) in the context of the class member functions.

Constructors

Constructor	Description
hash_set	Constructs a `hash_set` that is empty or that is a copy of all or part of some other `hash_set`.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for the `hash_set` object.
const_iterator	A type that provides a bidirectional iterator that can read a **`const`** element in the `hash_set`.
const_pointer	A type that provides a pointer to a **`const`** element in a `hash_set`.
const_reference	A type that provides a reference to a **`const`** element stored in a `hash_set` for reading and performing **`const`** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **`const`** element in the `hash_set`.
difference_type	A signed integer type that can be used to represent the number of elements of a `hash_set` in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a `hash_set`.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `hash_set`.
key_type	A type that describes an object stored as an element of a `hash_set` in its capacity as sort key.
pointer	A type that provides a pointer to an element in a `hash_set`.
reference	A type that provides a reference to an element stored in a `hash_set`.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `hash_set`.
size_type	An unsigned integer type that can represent the number of elements in a `hash_set`.

Type name	Description
value_compare	A type that provides two function objects, a binary predicate of class compare that can compare two element values of a `hash_set` to determine their relative order and a unary predicate that hashes the elements.
value_type	A type that describes an object stored as an element of a `hash_set` in its capacity as a value.

Member functions

Member function	Description
begin	Returns an iterator that addresses the first element in the `hash_set`.
cbegin	Returns a const iterator addressing the first element in the `hash_set`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `hash_set`.
clear	Erases all the elements of a `hash_set`.
count	Returns the number of elements in a `hash_set` whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed `hash_set`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `hash_set`.
emplace	Inserts an element constructed in place into a `hash_set`.
emplace_hint	Inserts an element constructed in place into a `hash_set`, with a placement hint.
empty	Tests if a `hash_set` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `hash_set`.
equal_range	Returns a pair of iterators respectively to the first element in a `hash_set` with a key that is greater than a specified key and to the first element in the `hash_set` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a `hash_set` from specified positions or removes elements that match a specified key.

Member function	Description
find	Returns an iterator addressing the location of an element in a `hash_set` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `hash_set`.
insert	Inserts an element or a range of elements into a `hash_set`.
key_comp	Retrieves a copy of the comparison object used to order keys in a `hash_set`.
lower_bound	Returns an iterator to the first element in a `hash_set` with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `hash_set`.
rbegin	Returns an iterator addressing the first element in a reversed `hash_set`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `hash_set`.
size	Returns the number of elements in the `hash_set`.
swap	Exchanges the elements of two `hash_set`s.
upper_bound	Returns an iterator to the first element in a `hash_set` that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a `hash_set`.

Operators

Operator	Description
hash_set::operator=	Replaces the elements of a `hash_set` with a copy of another `hash_set`.

Requirements

Header: <hash_set>

Namespace: stdext

hash_set::allocator_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that represents the allocator class for the hash_set object.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::allocator_type allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter *Allocator*.

For more information on *Allocator*, see the Remarks section of the hash_set Class topic.

Example

See example for get_allocator for an example that uses `allocator_type`.

hash_set::begin

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator that addresses the first element in the hash_set.

```
C++
```

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_set or the location succeeding an empty hash_set.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_set object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_set object can be modified.

Example

C++

```
// hash_set_begin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_Iter = hs1.begin( );
   cout << "The first element of hs1 is " << *hs1_Iter << endl;

   hs1_Iter = hs1.begin( );
   hs1.erase( hs1_Iter );

   // The following 2 lines would err because the iterator is const
   // hs1_cIter = hs1.begin( );
   // hs1.erase( hs1_cIter );

   hs1_cIter = hs1.begin( );
   cout << "The first element of hs1 is now " << *hs1_cIter << endl;
}
```

Output

```
The first element of hs1 is 1
The first element of hs1 is now 2
```

hash_set::cbegin

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a const iterator that addresses the first element in the hash_set.

```
C++
```

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the hash_set or the location succeeding an empty `hash_set`.

Remarks

With the return value of `cbegin`, the elements in the `hash_set` object cannot be modified.

Example

```
C++
```

```
// hash_set_cbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_cIter = hs1.cbegin( );
   cout << "The first element of hs1 is " << *hs1_cIter << endl;
}
```

Output

```
The first element of hs1 is 1
```

hash_set::cend

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a const iterator that addresses the location succeeding the last element in a hash_set.

C++

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a hash_set. If the `hash_set` is empty, then `hash_set::cend == hash_set::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_set`. The value returned by `cend` should not be dereferenced.

Example

C++

```
// hash_set_cend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_cIter;
```

```
   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_cIter = hs1.cend( );
   hs1_cIter--;
   cout << "The last element of hs1 is " << *hs1_cIter << endl;
}
```

Output

```
The last element of hs1 is 3
```

hash_set::clear

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Erases all the elements of a hash_set.

C++

```
void clear();
```

Remarks

Example

C++

```
// hash_set_clear.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 1 );
```

```
    hs1.insert( 2 );

    cout << "The size of the hash_set is initially " << hs1.size( )
         << "." << endl;

    hs1.clear( );
    cout << "The size of the hash_set after clearing is "
         << hs1.size( ) << "." << endl;
}
```

Output

```
The size of the hash_set is initially 2.
The size of the hash_set after clearing is 0.
```

hash_set::const_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a bidirectional iterator that can read a `const` element in the hash_set.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See example for begin for an example that uses `const_iterator`.

hash_set::const_pointer

> ⓘ Note

This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a pointer to a **`const`** element in a hash_set.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a **`const`** hash_set object.

hash_set::const_reference

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a reference to a **`const`** element stored in a hash_set for reading and performing **`const`** operations.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_reference const_reference;
```

Remarks

Example

```
C++
```

```
// hash_set_const_ref.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 10 );
   hs1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *hs1.begin( );

   cout << "The first element in the hash_set is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the hash_set
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the hash_set is 10.
```

hash_set::const_reverse_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a bidirectional iterator that can read any `const` element in the hash_set.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_reverse_iterator const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_set in reverse.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`

hash_set::count

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns the number of elements in a hash_set whose key matches a parameter-specified key.

```
C++
```

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the hash_set.

Return Value

1 if the hash_set contains an element whose sort key matches the parameter key.

0 if the hash_set does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[lower_bound(*key*), upper_bound(*key*)).

Example

The following example demonstrates the use of the hash_set::count member function.

```
C++
```

```cpp
// hash_set_count.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1;
    hash_set<int>::size_type i;

    hs1.insert(1);
    hs1.insert(1);

    // Keys must be unique in hash_set, so duplicates are ignored.
    i = hs1.count(1);
    cout << "The number of elements in hs1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hs1.count(2);
    cout << "The number of elements in hs1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hs1 with a sort key of 1 is: 1.
The number of elements in hs1 with a sort key of 2 is: 0.
```

hash_set::crbegin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a const iterator addressing the first element in a reversed hash_set.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed hash_set or addressing what had been the last element in the unreversed `hash_set`.

Remarks

`crbegin` is used with a reversed hash_set just as hash_set::begin is used with a hash_set.

With the return value of `crbegin`, the `hash_set` object cannot be modified.

`crbegin` can be used to iterate through a `hash_set` backwards.

Example

C++

```
// hash_set_crbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_crIter = hs1.crbegin( );
   cout << "The first element in the reversed hash_set is "
        << *hs1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed hash_set is 30.
```

hash_set::crend

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_set.

```
C++
```

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_set (the location that had preceded the first element in the unreversed `hash_set`).

Remarks

`crend` is used with a reversed `hash_set` just as hash_set::end is used with a `hash_set`.

With the return value of `crend`, the `hash_set` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `hash_set`.

Example

```
C++
```

```
// hash_set_crend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_crIter = hs1.crend( );
   hs1_crIter--;
   cout << "The last element in the reversed hash_set is "
```

```
        << *hs1_crIter << "." << endl;
}
```

Output

```
The last element in the reversed hash_set is 10.
```

hash_set::difference_type

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A signed integer type that can be used to represent the number of elements of a hash_set in a range between elements pointed to by iterators.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random access container, such as vector or deque.

Example

C++

```
// hash_set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter, hs1_bIter, hs1_eIter;

   hs1.insert( 20 );
   hs1.insert( 10 );
   hs1.insert( 20 );   // Won't insert as hash_set elements are unique

   hs1_bIter = hs1.begin( );
   hs1_eIter = hs1.end( );

   hash_set <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( hs1_bIter, hs1_eIter, 5 );
   df_typ10 = count( hs1_bIter, hs1_eIter, 10 );
   df_typ20 = count( hs1_bIter, hs1_eIter, 20 );

   // The keys, and hence the elements, of a hash_set are unique,
   // so there is at most one of a given value
   cout << "The number '5' occurs " << df_typ5
        << " times in hash_set hs1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in hash_set hs1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in hash_set hs1.\n";

   // Count the number of elements in a hash_set
   hash_set <int>::difference_type  df_count = 0;
   hs1_Iter = hs1.begin( );
   while ( hs1_Iter != hs1_eIter)
   {
      df_count++;
      hs1_Iter++;
   }

   cout << "The number of elements in the hash_set hs1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in hash_set hs1.
The number '10' occurs 1 times in hash_set hs1.
The number '20' occurs 1 times in hash_set hs1.
The number of elements in the hash_set hs1 is: 2.
```

hash_set::emplace

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Inserts an element constructed in place into a hash_set.

C++

```
template <class ValTy>
pair <iterator, bool>
emplace(
    ValTy&& val);
```

Parameters

val
The value of an element to be inserted into the hash_set unless the `hash_set` already contains that element or, more generally, an element whose key is equivalently ordered.

Return Value

The `emplace` member function returns a pair whose `bool` component returns `true` if an insertion was make and `false` if the `hash_set` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

Remarks

Example

C++

```cpp
// hash_set_emplace.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set<string> hs3;
   string str1("a");

   hs3.emplace(move(str1));
   cout << "After the emplace insertion, hs3 contains "
      << *hs3.begin() << "." << endl;
}
```

Output

```
After the emplace insertion, hs3 contains a.
```

hash_set::emplace_hint

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Inserts an element constructed in place into a hash_set.

C++

```cpp
template <class ValTy>
iterator emplace(
    const_iterator _Where,
    ValTy&& val);
```

Parameters

val
The value of an element to be inserted into the hash_set unless the `hash_set` already contains that element or, more generally, an element whose key is equivalently ordered.

_Where
The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.)

Return Value

The hash_set::emplace member function returns an iterator that points to the position where the new element was inserted into the `hash_set`, or where the existing element with equivalent ordering is located.

Remarks

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *_Where*.

Example

C++

```
// hash_set_emplace_hint.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set<string> hs3;
   string str1("a");

   hs3.insert(hs3.begin(), move(str1));
   cout << "After the emplace insertion, hs3 contains "
      << *hs3.begin() << "." << endl;
}
```

Output

```
After the emplace insertion, hs3 contains a.
```

hash_set::empty

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Tests if a hash_set is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the hash_set is empty; `false` if the hash_set is nonempty.

Remarks

Example

C++

```
// hash_set_empty.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2;
   hs1.insert ( 1 );

   if ( hs1.empty( ) )
      cout << "The hash_set hs1 is empty." << endl;
   else
      cout << "The hash_set hs1 is not empty." << endl;

   if ( hs2.empty( ) )
      cout << "The hash_set hs2 is empty." << endl;
   else
      cout << "The hash_set hs2 is not empty." << endl;
}
```

Output

```
The hash_set hs1 is not empty.
The hash_set hs2 is empty.
```

hash_set::end

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator that addresses the location succeeding the last element in a hash_set.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_set. If the hash_set is empty, then hash_set::end == hash_set::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_set. The value returned by `end` should not be dereferenced.

Example

C++

```
// hash_set_end.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
```

```
   hash_set <int> hs1;
   hash_set <int> :: iterator hs1_Iter;
   hash_set <int> :: const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_Iter = hs1.end( );
   hs1_Iter--;
   cout << "The last element of hs1 is " << *hs1_Iter << endl;

   hs1.erase( hs1_Iter );

   // The following 3 lines would err because the iterator is const:
   // hs1_cIter = hs1.end( );
   // hs1_cIter--;
   // hs1.erase( hs1_cIter );

   hs1_cIter = hs1.end( );
   hs1_cIter--;
   cout << "The last element of hs1 is now " << *hs1_cIter << endl;
}
```

Output

```
The last element of hs1 is 3
The last element of hs1 is now 2
```

hash_set::equal_range

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a pair of iterators respectively to the first element in a hash set with a key that is equal to a specified key and to the first element in the hash set with a key that is greater than the key.

C++

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair pr returned by the member function, use `pr`. **first**, and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second**, and to dereference the upper bound iterator, use *(`pr`. **second**).

Remarks

Example

C++

```
// hash_set_equal_range.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_set<int> IntHSet;
   IntHSet hs1;
   hash_set <int> :: const_iterator hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
   p1 = hs1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the hash_set hs1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
```

```
        << "a key of 20 in the hash_set hs1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   hs1_RcIter = hs1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *hs1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = hs1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == hs1.end( ) ) && ( p2.second == hs1.end( ) ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key greater than or equal to 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key >= 40 is: "
           << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the hash_set hs1 is: 30.
The lower bound of the element with a key of 20 in the hash_set hs1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The hash_set hs1 doesn't have an element with a key greater than or equal to
40.
```

hash_set::erase

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Removes an element or a range of elements in a hash_set from specified positions or removes elements that match a specified key.

C++

```
iterator erase(iterator _Where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

_Where
Position of the element to be removed from the hash_set.

first
Position of the first element removed from the hash_set.

last
Position just beyond the last element removed from the hash_set.

key
The key of the elements to be removed from the hash_set.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_set if no such element exists. For the third member function, the number of elements that have been removed from the hash_set.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the hash_set::erase member function.

C++

```
// hash_set_erase.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_set<int> hs1, hs2, hs3;
    hash_set<int>::iterator pIter, Iter1, Iter2;
    int i;
    hash_set<int>::size_type n;

    for (i = 1; i < 5; i++)
```

```
    {
        hs1.insert (i);
        hs2.insert (i * i);
        hs3.insert (i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hs1.begin();
    hs1.erase(Iter1);

    cout << "After the 2nd element is deleted, the hash_set hs1 is:";
    for (pIter = hs1.begin(); pIter != hs1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [ first,  last)
    Iter1 = ++hs2.begin();
    Iter2 = --hs2.end();
    hs2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted, "
         << "the hash_set hs2 is:";
    for (pIter = hs2.begin(); pIter != hs2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given  key
    n = hs3.erase(2);

    cout << "After the element with a key of 2 is deleted, "
         << "the hash_set hs3 is:";
    for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hs3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hs3.begin();
    hs3.erase(Iter1);

    cout << "After another element (unique for hash_set) with a key "
         << endl;
    cout  << "equal to that of the 2nd element is deleted, "
          << "the hash_set hs3 is:";
    for (pIter = hs3.begin(); pIter != hs3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted, the hash_set hs1 is: 1 3 4.
After the middle two elements are deleted, the hash_set hs2 is: 16 4.
After the element with a key of 2 is deleted, the hash_set hs3 is: 0 1 3.
The number of elements removed from hs3 is: 1.
After another element (unique for hash_set) with a key
equal to that of the 2nd element is deleted, the hash_set hs3 is: 0 3.
```

hash_set::find

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.

C++

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key
The argument key to be matched by the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_set whose sort key is `equivalent` to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the hash_set object cannot be modified. If the return value of `find` is assigned to an `iterator`, the hash_set object can be modified.

Example

C++

```
// hash_set_find.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.find( 20 );
   cout << "The element of hash_set hs1 with a key of 20 is: "
        << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key of 40 is: "
           << *hs1_RcIter << "." << endl;

   // The element at a specific location in the hash_set can be found
   // by using a dereferenced iterator addressing the location
   hs1_AcIter = hs1.end( );
   hs1_AcIter--;
   hs1_RcIter = hs1.find( *hs1_AcIter );
   cout << "The element of hs1 with a key matching "
        << "that of the last element is: "
        << *hs1_RcIter << "." << endl;
}
```

Output

```
The element of hash_set hs1 with a key of 20 is: 20.
The hash_set hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

hash_set::get_allocator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns a copy of the allocator object used to construct the hash_set.

C++

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_set to manage memory, which is the template parameter *Allocator*.

For more information on *Allocator*, see the Remarks section of the hash_set Class topic.

Remarks

Allocators for the hash_set class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// hash_set_get_allocator.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
```

```
   using namespace std;
   using namespace stdext;

   // The following lines declare objects
   // that use the default allocator.
   hash_set <int, hash_compare <int, less<int> > > hs1;
   hash_set <int,  hash_compare <int, greater<int> > > hs2;
   hash_set <double, hash_compare <double,
      less<double> >, allocator<double> > hs3;

   hash_set <int, hash_compare <int,
      greater<int> > >::allocator_type hs2_Alloc;
   hash_set <double>::allocator_type hs3_Alloc;
   hs2_Alloc = hs2.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs1.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hs3.max_size( ) <<  "." << endl;

   // The following lines create a hash_set hs4
   // with the allocator of hash_set hs1.
   hash_set <int>::allocator_type hs4_Alloc;
   hash_set <int> hs4;
   hs4_Alloc = hs2.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hs2_Alloc == hs4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

hash_set::hash_set

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Constructs a `hash_set` that is empty or that is a copy of all or part of some other `hash_set`.

```
C++
```

```
hash_set();

explicit hash_set(
    const Traits& Comp);

hash_set(
    const Traits& Comp,
    const Allocator& Al);

hash_set(
    const hash_set<Key, Traits, Allocator>& Right);

hash_set(
    hash_set&& Right);

hash_set(
    initializer_list<Type> IList);

hash_set(
    initializer_list<Type> IList,
    const Compare& Comp);

hash_set(
    initializer_list<value_type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
hash_set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

Al
The storage allocator class to be used for this `hash_set` object, which defaults to `Allocator`.

Comp
The comparison function of type `const Traits` used to order the elements in the `hash_set`, which defaults to `hash_compare`.

Right
The `hash_set` of which the constructed `hash_set` is to be a copy.

First
The position of the first element in the range of elements to be copied.

Last
The position of the first element beyond the range of elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the `hash_set` and that can later be returned by calling hash_set::get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_sets.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_set` and that can later be returned by calling hash_set::key_comp. For more information on `Traits` see the hash_set Class topic.

The first constructor creates an empty initial `hash_set` The second specifies the type of comparison function (`Comp`) to be used in establishing the order of the elements, and the third explicitly specifies the allocator type (`Al`) to be used. The key word **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth and fifth constructors specify a copy of the `hash_set` `Right`.

The last sixth, seventh, and eighth constructors use an initializer_list for the elements.

The last constructors copy the range [`First`, `Last`) of a `hash_set` with increasing explicitness in specifying the type of comparison function of class Traits and allocator.

The eighth constructor moves the `hash_set` `Right`.

The actual order of elements in a `hash_set` container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

hash_set::insert

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Inserts an element or a range of elements into a `hash_set`.

```
C++
```

```
pair<iterator, bool> insert(
    const value_type& Val);

iterator insert(
    iterator Where,
    const value_type& Val);

void insert(
    initializer_list<value_type> IList)
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);
```

Parameters

Val
The value of an element to be inserted into the `hash_set` unless the `hash_set` already contains that element or, more generally, an element whose key is equivalently ordered.

Where
The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `_Where`.)

First
The position of the first element to be copied from a `hash_set`.

Last
The position just beyond the last element to be copied from a `hash_set`.

IList
The initializer_list from which to copy the elements.

Return Value

The first `insert` member function returns a pair whose **`bool`** component returns **`true`** if an insertion was make and **`false`** if the `hash_set` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first` and to dereference it, use `*(pr.first)`. To access the **`bool`** component of a pair `pr` returned by this member function, use `pr.second`, and to dereference it, use `*(pr.second)`.

The second `insert` member function returns an iterator that points to the position where the new element was inserted into the `hash_set`.

Remarks

The third member function inserts the elements in an initializer_list.

The third member function inserts the sequence of element values into a `hash_set` corresponding to each element addressed by an iterator of in the range [`First`, `Last`) of a specified `hash_set`.

hash_set::iterator

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a bidirectional iterator that can read or modify any element in a hash_set.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::iterator iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin for an example of how to declare and use `iterator`.

hash_set::key_comp

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Retrieves a copy of the hash traits object used to hash and order element key values in a hash_set.

```
C++
```

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a hash_set uses to order its elements, which is the template parameter *Traits*.

For more information on *Traits* see the hash_set Class topic.

Remarks

The stored object defines the member function:

`bool operator( const Key& _xVal, const Key& _yVal );`

which returns **`true`** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter *Traits*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

C++

```
// hash_set_key_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int, hash_compare < int, less<int> > >hs1;
   hash_set<int, hash_compare < int, less<int> > >::key_compare kc1
          = hs1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of hs1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of hs1."
        << endl;
   }

   hash_set <int, hash_compare < int, greater<int> > > hs2;
   hash_set<int, hash_compare < int, greater<int> > >::key_compare
         kc2 = hs2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if(result2 == true)
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of hs2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of hs2."
           << endl;
```

```
    }
}
```

hash_set::key_compare

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the hash_set.

```
C++

typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the hash_set Class topic.

Note that both `key_compare` and value_compare are synonyms for the template parameter *Traits*. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

hash_set::key_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that describes an object stored as an element of a hash_set in its capacity as sort key.

```
C++
```

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

For more information on *Key*, see the Remarks section of the hash_set Class topic.

Note that both `key_type` and value_type are synonyms for the template parameter *Key*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

See the example for value_type for an example of how to declare and use `key_type`.

hash_set::lower_bound

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.

```
C++
```

```
const_iterator lower_bound(const Key& key) const;

iterator lower_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

Example

C++

```
// hash_set_lower_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.lower_bound( 20 );
   cout << "The element of hash_set hs1 with a key of 20 is: "
        << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_set hs1 with a key of 40 is: "
           << *hs1_RcIter << "." << endl;

   // An element at a specific location in the hash_set can be found
   // by using a dereferenced iterator that addresses the location
   hs1_AcIter = hs1.end( );
   hs1_AcIter--;
   hs1_RcIter = hs1.lower_bound( *hs1_AcIter );
   cout << "The element of hs1 with a key matching "
        << "that of the last element is: "
```

```
        << *hs1_RcIter << "." << endl;
}
```

Output

```
The element of hash_set hs1 with a key of 20 is: 20.
The hash_set hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

hash_set::max_size

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns the maximum length of the hash_set.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_set.

Remarks

Example

C++

```
// hash_set_max_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::size_type i;
```

```
   i = hs1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_set is " << i << "." << endl;
}
```

hash_set::operator=

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Replaces the elements of the hash_set with a copy of another hash_set.

C++

```
hash_set& operator=(const hash_set& right);

hash_set& operator=(hash_set&& right);
```

Parameters

right
The hash_set being copied into the `hash_set`.

Remarks

After erasing any existing elements in a `hash_set`, `operator=` either copies or moves the contents of *right* into the `hash_set`.

Example

C++

```
// hash_set_operator_as.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set<int> v1, v2, v3;
```

```
    hash_set<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
       cout << iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << iter << " ";
    cout << endl;

// move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
       cout << iter << " ";
    cout << endl;
}
```

hash_set::pointer

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a pointer to an element in a hash_set.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a hash_set object.

hash_set::rbegin

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator addressing the first element in a reversed hash_set.

```
C++
```

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_set or addressing what had been the last element in the unreversed hash_set.

Remarks

`rbegin` is used with a reversed hash_set just as begin is used with a hash_set.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_set object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_set object can be modified.

`rbegin` can be used to iterate through a hash_set backwards.

Example

```
C++
```

```
// hash_set_rbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::reverse_iterator hs1_rIter;

   hs1.insert( 10 );
```

```
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_rIter = hs1.rbegin( );
   cout << "The first element in the reversed hash_set is "
        << *hs1_rIter << "." << endl;

   // begin can be used to start an iteration
   // through a hash_set in a forward order
   cout << "The hash_set is: ";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( );
         hs1_Iter++ )
      cout << *hs1_Iter << " ";
   cout << endl;

   // rbegin can be used to start an iteration
   // through a hash_set in a reverse order
   cout << "The reversed hash_set is: ";
   for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
         hs1_rIter++ )
      cout << *hs1_rIter << " ";
   cout << endl;

   // A hash_set element can be erased by dereferencing to its key
   hs1_rIter = hs1.rbegin( );
   hs1.erase ( *hs1_rIter );

   hs1_rIter = hs1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed hash_set is "<< *hs1_rIter << "."
        << endl;
}
```

Output

```
The first element in the reversed hash_set is 30.
The hash_set is: 10 20 30
The reversed hash_set is: 30 20 10
After the erasure, the first element in the reversed hash_set is 20.
```

hash_set::reference

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a reference to an element stored in a hash_set.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::reference reference;
```

Remarks

Example

C++

```
// hash_set_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;

   hs1.insert( 10 );
   hs1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *hs1.begin( );

   cout << "The first element in the hash_set is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the hash_set can be changed
   // by operating on its (non-const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the hash_set is now "
        << *hs1.begin() << "." << endl;
}
```

Output

```
The first element in the hash_set is 10.
The first element in the hash_set is now 15.
```

hash_set::rend

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_set (the location that had preceded the first element in the unreversed hash_set).

Remarks

`rend` is used with a reversed hash_set just as end is used with a hash_set.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the hash_set object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the hash_set object can be modified. The value returned by `rend` should not be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its hash_set.

Example

C++

```
// hash_set_rend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;
   hash_set <int>::reverse_iterator hs1_rIter;
   hash_set <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_rIter = hs1.rend( );
   hs1_rIter--;
   cout << "The last element in the reversed hash_set is "
        << *hs1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // through a hash_set in a forward order
   cout << "The hash_set is: ";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( );
         hs1_Iter++ )
      cout << *hs1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // through a hash_set in a reverse order
   cout << "The reversed hash_set is: ";
   for ( hs1_rIter = hs1.rbegin( ) ; hs1_rIter != hs1.rend( );
         hs1_rIter++ )
      cout << *hs1_rIter << " ";
   cout << "." << endl;

   hs1_rIter = hs1.rend( );
   hs1_rIter--;
   hs1.erase ( *hs1_rIter );

   hs1_rIter = hs1.rend( );
   hs1_rIter--;
   cout << "After the erasure, the last element in the "
        << "reversed hash_set is " << *hs1_rIter << "."
        << endl;
}
```

Output

```
The last element in the reversed hash_set is 10.
The hash_set is: 10 20 30 .
The reversed hash_set is: 30 20 10 .
After the erasure, the last element in the reversed hash_set is 20.
```

hash_set::reverse_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_set.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::reverse_iterator reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the hash_set in reverse.

Example

See the example for rbegin for an example of how to declare and use `reverse_iterator`.

hash_set::size

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns the number of elements in the hash_set.

```
C++
```

```
size_type size() const;
```

Return Value

The current length of the hash_set.

Remarks

Example

C++

```
// hash_set_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: size_type i;

   hs1.insert( 1 );
   i = hs1.size( );
   cout << "The hash_set length is " << i << "." << endl;

   hs1.insert( 2 );
   i = hs1.size( );
   cout << "The hash_set length is now " << i << "." << endl;
}
```

Output

```
The hash_set length is 1.
The hash_set length is now 2.
```

hash_set::size_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

An unsigned integer type that can represent the number of elements in a hash_set.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See the example for size for an example of how to declare and use `size_type`

hash_set::swap

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Exchanges the elements of two hash_sets.

C++

```
void swap(hash_set& right);
```

Parameters

right
The argument hash_set providing the elements to be swapped with the target hash_set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_sets whose elements are being exchanged.

Example

C++

```
// hash_set_swap.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   hash_set <int>::iterator hs1_Iter;

   hs1.insert( 10 );
```

```
   hs1.insert( 20 );
   hs1.insert( 30 );
   hs2.insert( 100 );
   hs2.insert( 200 );
   hs3.insert( 300 );

   cout << "The original hash_set hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   hs1.swap( hs2 );

   cout << "After swapping with hs2, list hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hs1, hs3 );

   cout << "After swapping with hs3, list hs1 is:";
   for ( hs1_Iter = hs1.begin( ); hs1_Iter != hs1.end( );
         hs1_Iter++ )
         cout << " " << *hs1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original hash_set hs1 is: 10 20 30.
After swapping with hs2, list hs1 is: 200 100.
After swapping with hs3, list hs1 is: 300.
```

hash_set::upper_bound

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Returns an iterator to the first element in a hash_set that with a key that is greater than a specified key.

C++

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a hash_set that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the hash_set if no match is found for the key.

Remarks

Example

C++

```
// hash_set_upper_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int> :: const_iterator hs1_AcIter, hs1_RcIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_RcIter = hs1.upper_bound( 20 );
   cout << "The first element of hash_set hs1 with a key greater "
        << "than 20 is: " << *hs1_RcIter << "." << endl;

   hs1_RcIter = hs1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
```

```
   if ( hs1_RcIter == hs1.end( ) )
      cout << "The hash_set hs1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of hash_set hs1 with a key > 40 is: "
           << *hs1_RcIter << "." << endl;

   // An element at a specific location in the hash_set can be found
   // by using a dereferenced iterator addressing the location
   hs1_AcIter = hs1.begin( );
   hs1_RcIter = hs1.upper_bound( *hs1_AcIter );
   cout << "The first element of hs1 with a key greater than "
        << endl << "that of the initial element of hs1 is: "
        << *hs1_RcIter << "." << endl;
}
```

Output

```
The first element of hash_set hs1 with a key greater than 20 is: 30.
The hash_set hs1 doesn't have an element with a key greater than 30.
The first element of hs1 with a key greater than
that of the initial element of hs1 is: 20.
```

hash_set::value_comp

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Retrieves a copy of the comparison object used to order element values in a hash_set.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a hash_set uses to order its elements, which is the template parameter *Compare*.

For more information on *Compare*, see the Remarks section of the hash_set Class topic.

Remarks

The stored object defines the member function:

`bool operator( const Key& _xVal, const Key& _yVal );`

which returns **`true`** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both value_compare and key_compare are synonyms for the template parameter *Compare*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

C++

```
// hash_set_value_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_set <int, hash_compare < int, less<int> > > hs1;
   hash_set <int, hash_compare < int, less<int> >  >::value_compare
      vc1 = hs1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of hs1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of hs1."
           << endl;
   }

   hash_set <int, hash_compare < int, greater<int> > > hs2;
   hash_set<int, hash_compare < int, greater<int> > >::value_compare
      vc2 = hs2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of hs2."
           << endl;
```

```
    }
    else
    {
       cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of hs2."
            << endl;
    }
}
```

hash_set::value_compare

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_set to determine their relative order and a unary predicate that hashes the elements.

```
C++

typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the hash_set Class topic.

Note that both key_compare and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

See the example for value_comp for an example of how to declare and use `value_compare`.

hash_set::value_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

A type that describes an object stored as an element of a hash_set in its capacity as a value.

C++

```
typedef Key value_type;
```

Example

C++

```
// hash_set_value_type.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1;
   hash_set <int>::iterator hs1_Iter;

   hash_set <int> :: value_type hsvt_Int;   // Declare value_type
   hsvt_Int = 10;             // Initialize value_type

   hash_set <int> :: key_type hskt_Int;   // Declare key_type
   hskt_Int = 20;             // Initialize key_type

   hs1.insert( hsvt_Int );         // Insert value into hs1
   hs1.insert( hskt_Int );         // Insert key into hs1

   // A hash_set accepts key_types or value_types as elements
   cout << "The hash_set has elements:";
   for ( hs1_Iter = hs1.begin( ) ; hs1_Iter != hs1.end( ); hs1_Iter++)
      cout << " " << *hs1_Iter;
   cout << "." << endl;
}
```

Output

```
The hash_set has elements: 10 20.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

hash_multiset Class

Article • 08/03/2021

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

The container class hash_multiset is an extension of the C++ Standard Library and is used for the storage and fast retrieval of data from a collection in which the values of the elements contained serve as the key values and are not required to be unique.

Syntax

```
C++
```

```
template <class Key, class Traits =hash_compare<Key, less <Key>>, class
Allocator =allocator <Key>>
class hash_multiset
```

Parameters

Key
The element data type to be stored in the hash_multiset.

Traits
The type which includes two function objects, one of class compare that is a binary predicate able to compare two element values as sort keys to determine their relative order and a hash function that is a unary predicate mapping key values of the elements to unsigned integers of type `size_t`. This argument is optional, and the `hash_compare<Key, less<Key> >` is the default value.

Allocator
The type that represents the stored allocator object that encapsulates details about the hash_multiset's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

The hash_multiset is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Hashed, because its elements are grouped into buckets based on the value of a hash function applied to the key values of the elements.
- Unique in the sense that each of its elements must have a unique key. Because hash_multiset is also a simple associative container, its elements are also unique.
- A class template because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The main advantage of hashing over sorting is greater efficiency: a successful hashing performs insertions, deletions, and finds in constant average time as compared with a time proportional to the logarithm of the number of elements in the container for sorting techniques. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

The choice of container type should be based in general on the type of searching and inserting required by the application. Hashed associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient when used with a well-designed hash function, performing them in a time that is on average constant and not dependent on the number of elements in the container. A well-designed hash function produces a uniform distribution of hashed values and minimizes the number of collisions, where a collision is said to occur when distinct key values are mapped into the same hashed value. In the worst case, with the worst possible hash function, the number of operations is proportional to the number of elements in the sequence (linear time).

The hash_multiset should be the associative container of choice when the conditions associating the values with their keys are satisfies by the application. The elements of a hash_multiset may be multiple and serve as their own sort keys, so keys are not unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a hash_set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique keywords, then a hash_map would be an appropriate structure to contain this data. If instead the definitions were not unique, then a hash_multimap would be the container of choice.

The hash_multiset orders the sequence it controls by calling a stored hash traits object of type value_compare. This stored object may be accessed by calling the member function key_comp. Such a function object must behave the same as an object of class `hash_compare<Key, less<Key> >`. Specifically, for all values *Key* of type `Key`, the call `Trait(Key)` yields a distribution of values of type `size_t`.

In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*, *y*) is a function object that has two argument objects x and y and a return value of true or false. An ordering imposed on a hash_multiset is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*, *y*) and *f*(*y*, *x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

The actual order of elements in the controlled sequence depends on the hash function, the ordering function, and the current size of the hash table stored in the container object. You cannot determine the current size of the hash table, so you cannot in general predict the order of elements in the controlled sequence. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The iterator provided by the hash_multiset class is a bidirectional iterator, but the class member functions insert and hash_multiset have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own hash_multiset of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal hash_multiset of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`first`, `last`) in the context of the class member functions.

Constructors

Constructor	Description
hash_multiset	Constructs a `hash_multiset` that is empty or that is a copy of all or part of some other `hash_multiset`.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for the `hash_multiset` object.
const_iterator	A type that provides a bidirectional iterator that can read a **`const`** element in the `hash_multiset`.
const_pointer	A type that provides a pointer to a **`const`** element in a `hash_multiset`.
const_reference	A type that provides a reference to a **`const`** element stored in a `hash_multiset` for reading and performing **`const`** operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any **`const`** element in the `hash_multiset`.
difference_type	A signed integer type that provides the difference between two iterators that address elements within the same `hash_multiset`.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a `hash_multiset`.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `hash_multiset`.
key_type	A type that describes an object stored as an element of a `hash_set` in its capacity as sort key.
pointer	A type that provides a pointer to an element in a `hash_multiset`.
reference	A type that provides a reference to an element stored in a `hash_multiset`.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `hash_multiset`.
size_type	An unsigned integer type that can represent the number of elements in a `hash_multiset`.
value_compare	A type that provides two function objects, a binary predicate of class compare that can compare two element values of a `hash_multiset` to determine their relative order and a unary predicate that hashes the elements.

Type name	Description
value_type	A type that describes an object stored as an element of a `hash_multiset` in its capacity as a value.

Member functions

Member function	Description
begin	Returns an iterator that addresses the first element in the `hash_multiset`.
cbegin	Returns a const iterator addressing the first element in the `hash_multiset`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `hash_multiset`.
clear	Erases all the elements of a `hash_multiset`.
count	Returns the number of elements in a `hash_multiset` whose key matches a parameter-specified key
crbegin	Returns a const iterator addressing the first element in a reversed `hash_multiset`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `hash_multiset`.
emplace	Inserts an element constructed in place into a `hash_multiset`.
emplace_hint	Inserts an element constructed in place into a `hash_multiset`, with a placement hint.
empty	Tests if a `hash_multiset` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `hash_multiset`.
equal_range	Returns a pair of iterators respectively to the first element in a `hash_multiset` with a key that is greater than a specified key and to the first element in the `hash_multiset` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a `hash_multiset` from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a `hash_multiset` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `hash_multiset`.

Member function	Description
insert	Inserts an element or a range of elements into a `hash_multiset`.
key_comp	Retrieves a copy of the comparison object used to order keys in a `hash_multiset`.
lower_bound	Returns an iterator to the first element in a `hash_multiset` with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `hash_multiset`.
rbegin	Returns an iterator addressing the first element in a reversed `hash_multiset`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `hash_multiset`.
size	Returns the number of elements in the `hash_multiset`.
swap	Exchanges the elements of two `hash_multiset`s.
upper_bound	Returns an iterator to the first element in a `hash_multiset` that with a key that is equal to or greater than a specified key.
value_comp	Retrieves a copy of the hash traits object used to hash and order element key values in a `hash_multiset`.

Operators

Operator	Description
hash_multiset::operator=	Replaces the elements of the hash_multiset with a copy of another hash_multiset.

Requirements

Header: <hash_set>

Namespace: stdext

hash_multiset::allocator_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that represents the allocator class for the hash_multiset object.

```
C++

typedef list<typename Traits::value_type, typename
Traits::allocator_type>::allocator_type allocator_type;
```

Example

See example for get_allocator for an example using `allocator_type`

hash_multiset::begin

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator that addresses the first element in the hash_multiset.

```
C++

const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the hash_multiset or the location succeeding an empty hash_multiset.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the hash_multiset object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the hash_multiset object can be modified.

Example

```
C++
```

```cpp
// hash_multiset_begin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::const_iterator hms1_cIter;

   hms1.insert( 1 );
   hms1.insert( 2 );
   hms1.insert( 3 );

   hms1_Iter = hms1.begin( );
   cout << "The first element of hms1 is " << *hms1_Iter << endl;

   hms1_Iter = hms1.begin( );
   hms1.erase( hms1_Iter );

   // The following 2 lines would err because the iterator is const
   // hms1_cIter = hms1.begin( );
   // hms1.erase( hms1_cIter );

   hms1_cIter = hms1.begin( );
   cout << "The first element of hms1 is now " << *hms1_cIter << endl;
}
```

Output

```
The first element of hms1 is 1
The first element of hms1 is now 2
```

hash_multiset::cbegin

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a const iterator that addresses the first element in the hash_multiset.

C++

```
const_iterator cbegin() const;
```

Return Value

A const bidirectional iterator addressing the first element in the hash_multiset or the location succeeding an empty `hash_multiset`.

Remarks

With the return value of `cbegin`, the elements in the `hash_multiset` object cannot be modified.

Example

C++

```
// hash_multiset_cbegin.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1;
   hash_multiset <int>::const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_cIter = hs1.cbegin( );
   cout << "The first element of hs1 is " << *hs1_cIter << endl;
}
```

Output

```
The first element of hs1 is 1
```

hash_multiset::cend

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a const iterator that addresses the location succeeding the last element in a hash_multiset.

C++

```
const_iterator cend() const;
```

Return Value

A const bidirectional iterator that addresses the location succeeding the last element in a hash_multiset. If the `hash_multiset` is empty, then `hash_multiset::cend == hash_multiset::begin`.

Remarks

`cend` is used to test whether an iterator has reached the end of its `hash_multiset`. The value returned by `cend` should not be dereferenced.

Example

C++

```
// hash_multiset_cend.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1;
   hash_multiset <int> :: const_iterator hs1_cIter;

   hs1.insert( 1 );
   hs1.insert( 2 );
   hs1.insert( 3 );

   hs1_cIter = hs1.cend( );
   hs1_cIter--;
   cout << "The last element of hs1 is " << *hs1_cIter << endl;
}
```

Output

```
The last element of hs1 is 3
```

hash_multiset::clear

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Erases all the elements of a hash_multiset.

C++

```
void clear();
```

Remarks

Example

C++

```
// hash_multiset_clear.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 1 );
   hms1.insert( 2 );

   cout << "The size of the hash_multiset is initially " << hms1.size( )
        << "." << endl;

   hms1.clear( );
   cout << "The size of the hash_multiset after clearing is "
        << hms1.size( ) << "." << endl;
}
```

Output

```
The size of the hash_multiset is initially 2.
The size of the hash_multiset after clearing is 0.
```

hash_multiset::const_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a bidirectional iterator that can read a **`const`** element in the hash_multiset.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_iterator const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See example for begin for an example using `const_iterator`.

hash_multiset::const_pointer

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a pointer to a **`const`** element in a hash_multiset.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a **`const`** hash_multiset object.

hash_multiset::const_reference

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a reference to a **`const`** element stored in a hash_multiset for reading and performing **`const`** operations.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::const_reference const_reference;
```

Remarks

Example

C++

```
// hash_multiset_const_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 10 );
   hms1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *hms1.begin( );
```

```
   cout << "The first element in the hash_multiset is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the hash_multiset
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the hash_multiset is 10.
```

hash_multiset::const_reverse_iterator

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a bidirectional iterator that can read any `const` element in the hash_multiset.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::const_reverse_iterator const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is use to iterate through the hash_multiset in reverse.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

hash_multiset::count

> ⓘ Note

This API is obsolete. The alternative is **unordered_multiset Class**.

Returns the number of elements in a hash_multiset whose key matches a parameter-specified key.

```
C++
```

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the hash_multiset.

Return Value

The number of elements in the hash_multiset with the parameter-specified key.

Remarks

The member function returns the number of elements in the following range:

[lower_bound(*key*), upper_bound(*key*)).

Example

The following example demonstrates the use of the hash_multiset::count member function.

```
C++
```

```
// hash_multiset_count.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1;
    hash_multiset<int>::size_type i;

    hms1.insert(1);
```

```
    hms1.insert(1);

    // Keys do not need to be unique in hash_multiset,
    // so duplicates may exist.
    i = hms1.count(1);
    cout << "The number of elements in hms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = hms1.count(2);
    cout << "The number of elements in hms1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in hms1 with a sort key of 1 is: 2.
The number of elements in hms1 with a sort key of 2 is: 0.
```

hash_multiset::crbegin

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a const iterator addressing the first element in a reversed hash_multiset.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed hash_multiset or addressing what had been the last element in the unreversed `hash_multiset`.

Remarks

`crbegin` is used with a reversed `hash_multiset` just as hash_multiset::begin is used with a `hash_multiset`.

With the return value of `crbegin`, the `hash_multiset` object cannot be modified.

`crbegin` can be used to iterate through a `hash_multiset` backwards.

Example

C++

```
// hash_multiset_crbegin.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1;
   hash_multiset <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_crIter = hs1.crbegin( );
   cout << "The first element in the reversed hash_multiset is "
        << *hs1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed hash_multiset is 30.
```

hash_multiset::crend

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a const iterator that addresses the location succeeding the last element in a reversed hash_multiset.

C++

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multiset (the location that had preceded the first element in the unreversed `hash_multiset`).

Remarks

`crend` is used with a reversed `hash_multiset` just as hash_multiset::end is used with a `hash_multiset`.

With the return value of `crend`, the `hash_multiset` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its hash_multiset.

Example

C++

```
// hash_multiset_crend.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1;
   hash_multiset <int>::const_reverse_iterator hs1_crIter;

   hs1.insert( 10 );
   hs1.insert( 20 );
   hs1.insert( 30 );

   hs1_crIter = hs1.crend( );
   hs1_crIter--;
   cout << "The last element in the reversed hash_multiset is "
        << *hs1_crIter << "." << endl;
}
```

Output

```
The last element in the reversed hash_multiset is 10.
```

hash_multiset::difference_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A signed integer type that provides the difference between two iterators that address elements within the same hash_multiset.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set. Subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector or deque.

Example

C++

```
// hash_multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <hash_set>
#include <algorithm>

int main( )
{
   using namespace std;
   using namespace stdext;
```

```
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter, hms1_bIter, hms1_eIter;

   hms1.insert( 20 );
   hms1.insert( 10 );

   // hash_multiset elements need not be unique
   hms1.insert( 20 );

   hms1_bIter = hms1.begin( );
   hms1_eIter = hms1.end( );

   hash_multiset <int>::difference_type   df_typ5, df_typ10,
        df_typ20;

   df_typ5 = count( hms1_bIter, hms1_eIter, 5 );
   df_typ10 = count( hms1_bIter, hms1_eIter, 10 );
   df_typ20 = count( hms1_bIter, hms1_eIter, 20 );

   // The keys & hence the elements of a hash_multiset
   // need not be unique and may occur multiple times
   cout << "The number '5' occurs " << df_typ5
        << " times in hash_multiset hms1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in hash_multiset hms1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in hash_multiset hms1.\n";

   // Count the number of elements in a hash_multiset
   hash_multiset <int>::difference_type  df_count = 0;
   hms1_Iter = hms1.begin( );
   while ( hms1_Iter != hms1_eIter)
   {
      df_count++;
      hms1_Iter++;
   }

   cout << "The number of elements in the hash_multiset hms1 is "
        << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in hash_multiset hms1.
The number '10' occurs 1 times in hash_multiset hms1.
The number '20' occurs 2 times in hash_multiset hms1.
The number of elements in the hash_multiset hms1 is 3.
```

hash_multiset::emplace

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Inserts an element constructed in place into a hash_multiset.

C++

```
template <class ValTy>
iterator insert(ValTy&& val);
```

Parameters

val
The value of an element to be inserted into the hash_multiset unless the `hash_multiset` already contains that element or, more generally, an element whose key is equivalently ordered.

Return Value

The `emplace` member function returns an iterator that points to the position where the new element was inserted.

Remarks

Example

C++

```
// hash_multiset_emplace.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset<string> hms3;
   string str1("a");

   hms3.emplace(move(str1));
```

```
    cout << "After the emplace insertion, hms3 contains "
       << *hms3.begin() << "." << endl;
}
```

Output

```
After the emplace insertion, hms3 contains a.
```

hash_multiset::emplace_hint

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Inserts an element constructed in place into a hash_multiset, with a placement hint.

C++

```
template <class ValTy>
iterator insert(
    const_iterator where,
    ValTy&& val);
```

Parameters

val
The value of an element to be inserted into the hash_multiset unless the `hash_multiset` already contains that element or, more generally, an element whose key is equivalently ordered.

where
The place to start searching for the correct point of insertion. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *where*.)

Return Value

The hash_multiset::emplace member function returns an iterator that points to the position where the new element was inserted into the `hash_multiset`.

Remarks

Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows *where*.

Example

C++

```
// hash_multiset_emplace_hint.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset<string> hms1;
   string str1("a");

   hms1.insert(hms1.begin(), move(str1));
   cout << "After the emplace insertion, hms1 contains "
      << *hms1.begin() << "." << endl;
}
```

Output

```
After the emplace insertion, hms1 contains a.
```

hash_multiset::empty

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Tests if a hash_multiset is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the hash_multiset is empty; `false` if the hash_multiset is nonempty.

Remarks

Example

C++

```
// hash_multiset_empty.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1, hms2;
   hms1.insert ( 1 );

   if ( hms1.empty( ) )
      cout << "The hash_multiset hms1 is empty." << endl;
   else
      cout << "The hash_multiset hms1 is not empty." << endl;

   if ( hms2.empty( ) )
      cout << "The hash_multiset hms2 is empty." << endl;
   else
      cout << "The hash_multiset hms2 is not empty." << endl;
}
```

Output

```
The hash_multiset hms1 is not empty.
The hash_multiset hms2 is empty.
```

hash_multiset::end

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator that addresses the location succeeding the last element in a hash_multiset.

```
C++
```

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a hash_multiset. If the hash_multiset is empty, then hash_multiset::end == hash_multiset::begin.

Remarks

`end` is used to test whether an iterator has reached the end of its hash_multiset. The value returned by `end` should not be dereferenced.

Example

```
C++
```

```
// hash_multiset_end.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: iterator hms1_Iter;
   hash_multiset <int> :: const_iterator hms1_cIter;

   hms1.insert( 1 );
   hms1.insert( 2 );
   hms1.insert( 3 );

   hms1_Iter = hms1.end( );
   hms1_Iter--;
   cout << "The last element of hms1 is " << *hms1_Iter << endl;

   hms1.erase( hms1_Iter );
```

```
    // The following 3 lines would err because the iterator is const
    // hms1_cIter = hms1.end( );
    // hms1_cIter--;
    // hms1.erase( hms1_cIter );

    hms1_cIter = hms1.end( );
    hms1_cIter--;
    cout << "The last element of hms1 is now " << *hms1_cIter << endl;
}
```

Output

```
The last element of hms1 is 3
The last element of hms1 is now 2
```

hash_multiset::equal_range

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a pair of iterators respectively to the first element in a hash_multiset with a key that is greater than a specified key and to the first element in the hash_multiset with a key that is equal to or greater than the key.

C++

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **first** and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second** and to dereference the upper bound iterator, use *(`pr`. **second**).

Example

C++

```
// hash_multiset_equal_range.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   typedef hash_multiset<int> IntHSet;
   IntHSet hms1;
   hash_multiset <int> :: const_iterator hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   pair <IntHSet::const_iterator, IntHSet::const_iterator> p1, p2;
   p1 = hms1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20\nin the hash_multiset hms1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20\nin the hash_multiset hms1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   hms1_RcIter = hms1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *hms1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = hms1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == hms1.end( ) )
      && ( p2.second == hms1.end( ) ) )
      cout << "The hash_multiset hms1 doesn't have an element "
```

```
            << "with a key less than 40." << endl;
   else
      cout << "The element of hash_multiset hms1"
           << "with a key >= 40 is: "
           << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20
in the hash_multiset hms1 is: 30.
The lower bound of the element with a key of 20
in the hash_multiset hms1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The hash_multiset hms1 doesn't have an element with a key less than 40.
```

hash_multiset::erase

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Removes an element or a range of elements in a hash_multiset from specified positions or removes elements that match a specified key.

C++

```
iterator erase(iterator where);

iterator erase(iterator first, iterator last);

size_type erase(const key_type& key);
```

Parameters

where
Position of the element to be removed from the hash_multiset.

first
Position of the first element removed from the hash_multiset.

last
Position just beyond the last element removed from the hash_multiset.

key
The key of the elements to be removed from the hash_multiset.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the hash_multiset if no such element exists. For the third member function, the number of elements that have been removed from the hash_multiset.

Remarks

The member functions never throw an exception.

Example

The following example demonstrates the use of the hash_multiset::erase member function.

C++

```
// hash_multiset_erase.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main()
{
    using namespace std;
    using namespace stdext;
    hash_multiset<int> hms1, hms2, hms3;
    hash_multiset<int> :: iterator pIter, Iter1, Iter2;
    int i;
    hash_multiset<int>::size_type n;

    for (i = 1; i < 5; i++)
    {
        hms1.insert(i);
        hms2.insert(i * i);
        hms3.insert(i - 1);
    }

    // The 1st member function removes an element at a given position
    Iter1 = ++hms1.begin();
    hms1.erase(Iter1);

    cout << "After the 2nd element is deleted,\n"
         << "the hash_multiset hms1 is:" ;
```

```
    for (pIter = hms1.begin(); pIter != hms1.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 2nd member function removes elements
    // in the range [ first,  last)
    Iter1 = ++hms2.begin();
    Iter2 = --hms2.end();
    hms2.erase(Iter1, Iter2);

    cout << "After the middle two elements are deleted,\n"
         << "the hash_multiset hms2 is:" ;
    for (pIter = hms2.begin(); pIter != hms2.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function removes elements with a given  key
    n = hms3.erase(2);

    cout << "After the element with a key of 2 is deleted,\n"
         << "the hash_multiset hms3 is:" ;
    for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;

    // The 3rd member function returns the number of elements removed
    cout << "The number of elements removed from hms3 is: "
         << n << "." << endl;

    // The dereferenced iterator can also be used to specify a key
    Iter1 = ++hms3.begin();
    hms3.erase(Iter1);

    cout << "After another element with a key "
         << "equal to that of the 2nd element\n"
         << "is deleted, the hash_multiset hms3 is:" ;
    for (pIter = hms3.begin(); pIter != hms3.end(); pIter++)
        cout << " " << *pIter;
    cout << "." << endl;
}
```

Output

```
After the 2nd element is deleted,
the hash_multiset hms1 is: 1 3 4.
After the middle two elements are deleted,
the hash_multiset hms2 is: 16 4.
After the element with a key of 2 is deleted,
the hash_multiset hms3 is: 0 1 3.
The number of elements removed from hms3 is: 1.
After another element with a key equal to that of the 2nd element
is deleted, the hash_multiset hms3 is: 0 3.
```

hash_multiset::find

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator addressing the location of an element in a hash_multiset that has a key equivalent to a specified key.

```
C++
```

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key
The argument key to be matched by the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of an element equivalent to a specified key or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

The member function returns an iterator that addresses an element in the hash_multiset whose sort key is `equivalent` to the argument key under a binary predicate that induces an ordering based on a less-than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the hash_multiset object cannot be modified. If the return value of `find` is assigned to an `iterator`, the hash_multiset object can be modified.

Example

```
C++
```

```
// hash_multiset_find.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.find( 20 );
   cout << "The element of hash_multiset hms1 with a key of 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.find( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_multiset hms1 with a key of 40 is: "
           << *hms1_RcIter << "." << endl;

   // The element at a specific location in the hash_multiset can be found
   // by using a dereferenced iterator addressing the location
   hms1_AcIter = hms1.end( );
   hms1_AcIter--;
   hms1_RcIter = hms1.find( *hms1_AcIter );
   cout << "The element of hms1 with a key matching "
        << "that of the last element is: "
        << *hms1_RcIter << "." << endl;
}
```

Output

```
The element of hash_multiset hms1 with a key of 20 is: 20.
The hash_multiset hms1 doesn't have an element with a key of 40.
The element of hms1 with a key matching that of the last element is: 30.
```

hash_multiset::get_allocator

ⓘ Note

This API is obsolete. The alternative is **unordered_multiset Class**.

Returns a copy of the allocator object used to construct the hash_multiset.

```
C++
```

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the hash_multiset to manage memory, which is the class's template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the hash_multiset Class topic.

Remarks

Allocators for the hash_multiset class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
C++
```

```
// hash_multiset_get_allocator.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   // The following lines declare objects
   // that use the default allocator.
   hash_multiset <int, hash_compare <int, less<int> > > hms1;
   hash_multiset <int, hash_compare <int, greater<int> > > hms2;
   hash_multiset <double, hash_compare <double,
      less<double> >, allocator<double> > hms3;

   hash_multiset <int, hash_compare <int,
```

```
      greater<int> > >::allocator_type hms2_Alloc;
   hash_multiset <double>::allocator_type hms3_Alloc;
   hms2_Alloc = hms2.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms1.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << hms3.max_size( ) <<  "." << endl;

   // The following lines create a hash_multiset hms4
   // with the allocator of hash_multiset hms1.
   hash_multiset <int>::allocator_type hms4_Alloc;
   hash_multiset <int> hms4;
   hms4_Alloc = hms2.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( hms2_Alloc == hms4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

hash_multiset::hash_multiset

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Constructs a `hash_multiset` that is empty or that is a copy of all or part of some other `hash_multiset`.

C++

```
hash_multiset();

explicit hash_multiset(
    const Traits& Comp);
```

```
hash_multiset(
    const Traits& Comp,
    const Allocator& Al);

hash_multiset(
    const hash_multiset<Key, Traits, Allocator>& Right);

hash_multiset(
    hash_multiset&& Right
};
hash_multiset (initializer_list<Type> IList);

hash_multiset(
    initializer_list<Tu[e> IList, const Compare& Comp):
hash_multiset(
    initializer_list<Type> IList, const Compare& Comp, const Allocator& Al);

template <class InputIterator>
hash_multiset(
    InputIterator first,
    InputIterator last);

template <class InputIterator>
hash_multiset(
    InputIterator first,
    InputIterator last,
    const Traits& Comp);

template <class InputIterator>
hash_multiset(
    InputIterator first,
    InputIterator last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

Al

The storage allocator class to be used for this `hash_multiset` object, which defaults to `Allocator`.

Comp

The comparison function of type `const Traits` used to order the elements in the `hash_multiset`, which defaults to `hash_compare`.

Right

The `hash_multiset` of which the constructed `hash_multiset` is to be a copy.

first
The position of the first element in the range of elements to be copied.

last
The position of the first element beyond the range of elements to be copied.

IList
The initializer_list that contains the elements to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the `hash_multiset` and that can later be returned by calling hash_multiset::get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their hash_multisets.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `hash_multiset` and that can later be returned by calling hash_multiset::key_comp. For more information on `Traits` see the hash_multiset Class topic.

The first three constructors specify an empty initial `hash_multiset`, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor moves the `hash_multiset` `Right`.

The fifth, sixth, and seventh constructors use an initializer_list.

The last three constructors copy the range [`first`, `last`) of a `hash_multiset` with increasing explicitness in specifying the type of comparison function of class Compare and allocator.

The actual order of elements in a hashed set container depends on the hash function, the ordering function and the current size of the hash table and cannot, in general, be predicted as it could with the set container, where it was determined by the ordering function alone.

hash_multiset::insert

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Inserts an element or a range of elements into a hash_multiset.

C++

```
iterator insert(
    const Type& value);

iterator insert(
    iterator where,
    const Type& Al);

void insert(
    initializer_list<Type> IList);

iterator insert(
    const Type& value);

iterator insert(
    Iterator where,
    const Type& value);

template <class InputIterator>
void insert(
    InputIterator first,
    InputIterator last);

template <class ValTy>
iterator insert(
    ValTy&& value);

template <class ValTy>
iterator insert(
    const_iterator where,
    ValTy&& value);
```

Parameters

value
The value of an element to be inserted into the hash_multiset unless the hash_multiset already contains that element or, more generally, an element whose key is equivalently ordered.

where
The place to start searching for the correct point of insertion. (Insertion can occur in

amortized constant time, instead of logarithmic time, if the insertion point immediately follows *where*.)

first
The position of the first element to be copied from a hash_multiset.

last
The position just beyond the last element to be copied from a hash_multiset.

IList
The initializer_list that contains the elements to copy.

Return Value

The first two insert member functions return an iterator that points to the position where the new element was inserted.

The next three member functions use an initializer_list.

The third member function inserts the sequence of element values into a hash_multiset corresponding to each element addressed by an iterator of in the range [`first`, `last`) of a specified hash_multiset.

Remarks

Insertion can occur in amortized constant time for the hint version of insert, instead of logarithmic time, if the insertion point immediately follows *where*.

hash_multiset::iterator

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a bidirectional iterator that can read or modify any element in a hash_multiset.

```
C++
```

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::iterator iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See example for begin for an example of how to declare and use `iterator`.

hash_multiset::key_comp

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Retrieves a copy of the comparison object used to order keys in a hash_multiset.

```
C++

key_compare key_comp() const;
```

Return Value

Returns the hash_multiset template parameter *Traits*, which contains function objects that are used to hash and to order the elements of the container.

For more information on *Traits* see the hash_multiset Class topic.

Remarks

The stored object defines a member function:

`bool operator<(const Key& _xVal, const Key& _yVal);`

which returns **`true`** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter *Traits*. Both types are provided for the hash_multiset and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

C++

```
// hash_multiset_key_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multiset <int, hash_compare < int, less<int> > >hms1;
   hash_multiset<int, hash_compare < int, less<int> > >::key_compare kc1
          = hms1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of hms1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of hms1."
        << endl;
   }

   hash_multiset <int, hash_compare < int, greater<int> > > hms2;
   hash_multiset<int, hash_compare < int, greater<int> > >::key_compare
         kc2 = hms2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of hms2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of hms2."
           << endl;
   }
}
```

hash_multiset::key_compare

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

```
C++
```

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the hash_multiset Class topic.

Note that both `key_compare` and value_compare are synonyms for the template parameter *Traits*. Both types are provided for the hash_set and hash_multiset classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

See example for key_comp for an example of how to declare and use `key_compare`.

hash_multiset::key_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the hash_multiset.

```
C++
```

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *Key*.

Note that both `key_type` and value_type are synonyms for the template parameter *Key*. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

For more information on *Key*, see the Remarks section of the hash_multiset Class topic.

Example

See example for value_type for an example of how to declare and use `key_type`.

hash_multiset::lower_bound

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator to the first element in a hash_multiset with a key that is equal to or greater than a specified key.

```
C++

const_iterator lower_bound(const Key& key) const;

iterator lower_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of the first element in a hash_multiset with a key that is equal to or greater than the argument key, or that

addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

Example

C++

```
// hash_multiset_lower_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main() {
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.lower_bound( 20 );
   cout << "The element of hash_multiset hms1 with a key of 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of hash_multiset hms1 with a key of 40 is: "
           << *hms1_RcIter << "." << endl;

   // An element at a specific location in the hash_multiset can be found
   // by using a dereferenced iterator that addresses the location
   hms1_AcIter = hms1.end( );
   hms1_AcIter--;
   hms1_RcIter = hms1.lower_bound( *hms1_AcIter );
   cout << "The element of hms1 with a key matching "
        << "that of the last element is: "
        << *hms1_RcIter << "." << endl;
}
```

hash_multiset::max_size

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns the maximum length of the hash_multiset.

```
C++
```

```
size_type max_size() const;
```

Return Value

The maximum possible length of the hash_multiset.

Remarks

Example

```
C++
```

```
// hash_multiset_max_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::size_type i;

   i = hms1.max_size( );
   cout << "The maximum possible length "
        << "of the hash_multiset is " << i << "." << endl;
}
```

hash_multiset::operator=

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Replaces the elements of the hash_multiset with a copy of another hash_multiset.

C++

```
hash_multiset& operator=(const hash_multiset& right);

hash_multiset& operator=(hash_multiset&& right);
```

Parameters

right
The hash_multiset being copied into the `hash_multiset`.

Remarks

After erasing any existing elements in a `hash_multiset`, `operator=` either copies or moves the contents of *right* into the `hash_multiset`.

Example

C++

```
// hash_multiset_operator_as.cpp
// compile with: /EHsc
#include <hash_multiset>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset<int> v1, v2, v3;
   hash_multiset<int>::iterator iter;

   v1.insert(10);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter << " ";
   cout << endl;
```

```
// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter << " ";
   cout << endl;
}
```

hash_multiset::pointer

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a pointer to an element in a hash_multiset.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a multiset object.

hash_multiset::rbegin

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator addressing the first element in a reversed hash_multiset.

C++

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed hash_multiset or addressing what had been the last element in the unreversed hash_multiset.

Remarks

`rbegin` is used with a reversed hash_multiset just as begin is used with a hash_multiset.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the hash_multiset object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the hash_multiset object can be modified.

`rbegin` can be used to iterate through a hash_multiset backwards.

Example

C++

```
// hash_multiset_rbegin.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::reverse_iterator hms1_rIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_rIter = hms1.rbegin( );
   cout << "The first element in the reversed hash_multiset is "
        << *hms1_rIter << "." << endl;

   // begin can be used to start an iteration
   // through a hash_multiset in a forward order
   cout << "The hash_multiset is: ";
   for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( );
         hms1_Iter++ )
      cout << *hms1_Iter << " ";
   cout << endl;

   // rbegin can be used to start an iteration
```

```
   // through a hash_multiset in a reverse order
   cout << "The reversed hash_multiset is: ";
   for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( );
         hms1_rIter++ )
      cout << *hms1_rIter << " ";
   cout << endl;

   // A hash_multiset element can be erased by dereferencing to its key
   hms1_rIter = hms1.rbegin( );
   hms1.erase ( *hms1_rIter );

   hms1_rIter = hms1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed hash_multiset is "<< *hms1_rIter << "."
        << endl;
}
```

Output

```
The first element in the reversed hash_multiset is 30.
The hash_multiset is: 10 20 30
The reversed hash_multiset is: 30 20 10
After the erasure, the first element in the reversed hash_multiset is 20.
```

hash_multiset::reference

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a reference to an element stored in a hash_multiset.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::reference reference;
```

Remarks

Example

C++

```cpp
// hash_multiset_reference.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;

   hms1.insert( 10 );
   hms1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   int &Ref1 = *hms1.begin( );

   cout << "The first element in the hash_multiset is "
        << Ref1 << "." << endl;

   // The value of the 1st element of the hash_multiset can be
   // changed by operating on its (non const) reference
   Ref1 = Ref1 + 5;

   cout << "The first element in the hash_multiset is now "
        << *hms1.begin() << "." << endl;
}
```

Output

```
The first element in the hash_multiset is 10.
The first element in the hash_multiset is now 15.
```

hash_multiset::rend

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator that addresses the location succeeding the last element in a reversed hash_multiset.

C++

```cpp
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed hash_multiset (the location that had preceded the first element in the unreversed hash_multiset).

Remarks

`rend` is used with a reversed hash_multiset just as end is used with a hash_multiset.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the hash_multiset object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the hash_multiset object can be modified. The value returned by `rend` should not be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its hash_multiset.

Example

C++

```
// hash_multiset_rend.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;
   hash_multiset <int>::reverse_iterator hms1_rIter;
   hash_multiset <int>::const_reverse_iterator hms1_crIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_rIter = hms1.rend( );
   hms1_rIter--;
   cout << "The last element in the reversed hash_multiset is "
        << *hms1_rIter << "." << endl;
```

```
   // end can be used to terminate an iteration
   // through a hash_multiset in a forward order
   cout << "The hash_multiset is: ";
   for ( hms1_Iter = hms1.begin( ) ; hms1_Iter != hms1.end( );
         hms1_Iter++ )
      cout << *hms1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // through a hash_multiset in a reverse order
   cout << "The reversed hash_multiset is: ";
   for ( hms1_rIter = hms1.rbegin( ) ; hms1_rIter != hms1.rend( );
         hms1_rIter++ )
      cout << *hms1_rIter << " ";
   cout << "." << endl;

   hms1_rIter = hms1.rend( );
   hms1_rIter--;
   hms1.erase ( *hms1_rIter );

   hms1_rIter = hms1.rend( );
   hms1_rIter--;
   cout << "After the erasure, the last element in the "
        << "reversed hash_multiset is " << *hms1_rIter << "."
        << endl;
}
```

Output

```
The last element in the reversed hash_multiset is 10.
The hash_multiset is: 10 20 30 .
The reversed hash_multiset is: 30 20 10 .
After the erasure, the last element in the reversed hash_multiset is 20.
```

hash_multiset::reverse_iterator

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides a bidirectional iterator that can read or modify an element in a reversed hash_multiset.

C++

```
typedef list<typename Traits::value_type, typename
Traits::allocator_type>::reverse_iterator reverse_iterator;
```

Remarks

A type `reverse_iterator` is use to iterate through the hash_multiset in reverse.

Example

See example for rbegin for an example of how to declare and use `reverse_iterator`.

hash_multiset::size

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns the number of elements in the hash_multiset.

C++

```
size_type size() const;
```

Return Value

The current length of the hash_multiset.

Remarks

Example

C++

```
// hash_multiset_size.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: size_type i;
```

```
    hms1.insert( 1 );
    i = hms1.size( );
    cout << "The hash_multiset length is " << i << "." << endl;

    hms1.insert( 2 );
    i = hms1.size( );
    cout << "The hash_multiset length is now " << i << "." << endl;
}
```

Output

```
The hash_multiset length is 1.
The hash_multiset length is now 2.
```

hash_multiset::size_type

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

An unsigned integer type that can represent the number of elements in a hash_multiset.

C++

```
typedef list<typename _Traits::value_type, typename
_Traits::allocator_type>::size_type size_type;
```

Remarks

Example

See example for size for an example of how to declare and use `size_type`

hash_multiset::swap

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Exchanges the elements of two hash_multisets.

C++

```
void swap(hash_multiset& right);
```

Parameters

right
The argument hash_multiset providing the elements to be swapped with the target hash_multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two hash_multisets whose elements are being exchanged.

Example

C++

```
// hash_multiset_swap.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1, hms2, hms3;
   hash_multiset <int>::iterator hms1_Iter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );
   hms2.insert( 100 );
   hms2.insert( 200 );
   hms3.insert( 300 );

   cout << "The original hash_multiset hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   hms1.swap( hms2 );
```

```
   cout << "After swapping with hms2, list hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( hms1, hms3 );

   cout << "After swapping with hms3, list hms1 is:";
   for ( hms1_Iter = hms1.begin( ); hms1_Iter != hms1.end( );
         hms1_Iter++ )
         cout << " " << *hms1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original hash_multiset hms1 is: 10 20 30.
After swapping with hms2, list hms1 is: 200 100.
After swapping with hms3, list hms1 is: 300.
```

hash_multiset::upper_bound

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Returns an iterator to the first element in a hash_multiset with a key that is greater than a specified key.

C++

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the hash_multiset being searched.

Return Value

An iterator or const_iterator that addresses the location of the first element in a hash_multiset with a key greater than the argument key, or that addresses the location succeeding the last element in the hash_multiset if no match is found for the key.

Remarks

Example

C++

```
// hash_multiset_upper_bound.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int> :: const_iterator hms1_AcIter, hms1_RcIter;

   hms1.insert( 10 );
   hms1.insert( 20 );
   hms1.insert( 30 );

   hms1_RcIter = hms1.upper_bound( 20 );
   cout << "The first element of hash_multiset hms1" << endl
        << "with a key greater than 20 is: "
        << *hms1_RcIter << "." << endl;

   hms1_RcIter = hms1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( hms1_RcIter == hms1.end( ) )
      cout << "The hash_multiset hms1 doesn't have an element\n"
           << "with a key greater than 30." << endl;
   else
      cout << "The element of hash_multiset hms1"
           << "with a key > 40 is: "
           << *hms1_RcIter << "." << endl;

   // An element at a specific location in the hash_multiset can be
   // found by using a dereferenced iterator addressing the location
   hms1_AcIter = hms1.begin( );
   hms1_RcIter = hms1.upper_bound( *hms1_AcIter );
   cout << "The first element of hms1 with a key greater than "
        << endl << "that of the initial element of hms1 is: "
```

```
        << *hms1_RcIter << "." << endl;
}
```

Output

```
The first element of hash_multiset hms1
with a key greater than 20 is: 30.
The hash_multiset hms1 doesn't have an element
with a key greater than 30.
The first element of hms1 with a key greater than
that of the initial element of hms1 is: 20.
```

hash_multiset::value_comp

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

Retrieves a copy of the comparison object used to order element values in a hash_multiset.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the hash_multiset template parameter *Traits*, which contains function objects that are used to hash and to order elements of the container.

For more information on *Traits* see the hash_multiset Class topic.

Remarks

The stored object defines a member function:

bool operator(**constKey&** `_xVal`, **const Key&** *_yVal*);

which returns **`true`** if `_xVal` precedes and is not equal to `_yVal` in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter *Traits*. Both types are provided for the hash_multiset and hash_multiset

classes, where they are identical, for compatibility with the hash_map and hash_multimap classes, where they are distinct.

Example

C++

```
// hash_multiset_value_comp.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;

   hash_multiset <int, hash_compare < int, less<int> > > hms1;
   hash_multiset <int, hash_compare < int, less<int> > >::value_compare
      vc1 = hms1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of hms1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of hms1."
           << endl;
   }

   hash_multiset <int, hash_compare < int, greater<int> > > hms2;
   hash_multiset<int, hash_compare < int, greater<int> > >::
           value_compare vc2 = hms2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of hms2."
           << endl;
   }
   else
   {
      cout << "vc2( 2,3 ) returns value of false, "
           << "where vc2 is the function object of hms2."
           << endl;
   }
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of hms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of hms2.
```

hash_multiset::value_compare

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that provides two function objects, a binary predicate of class compare that can compare two element values of a hash_multiset to determine their relative order and a unary predicate that hashes the elements.

C++

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter *Traits*.

For more information on *Traits* see the hash_multiset Class topic.

Note that both key_compare and `value_compare` are synonyms for the template parameter *Traits*. Both types are provided for the classes set and multiset, where they are identical, for compatibility with the classes map and multimap, where they are distinct.

Example

See example for value_comp for an example of how to declare and use `value_compare`.

hash_multiset::value_type

> **Note**
>
> This API is obsolete. The alternative is **unordered_multiset Class**.

A type that describes an object stored as an element as a hash_multiset in its capacity as a value.

C++

```
typedef Key value_type;
```

Example

C++

```
// hash_multiset_value_type.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hms1;
   hash_multiset <int>::iterator hms1_Iter;

   // Declare value_type
   hash_multiset <int> :: value_type hmsvt_Int;

   hmsvt_Int = 10;   // Initialize value_type

   // Declare key_type
   hash_multiset <int> :: key_type hmskt_Int;
   hmskt_Int = 20;             // Initialize key_type

   hms1.insert( hmsvt_Int );         // Insert value into s1
   hms1.insert( hmskt_Int );         // Insert key into s1

   // A hash_multiset accepts key_types or value_types as elements
   cout << "The hash_multiset has elements:";
   for ( hms1_Iter = hms1.begin() ; hms1_Iter != hms1.end( );
         hms1_Iter++)
      cout << " " << *hms1_Iter;
      cout << "." << endl;
}
```

Output

```
The hash_multiset has elements: 10 20.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<hash_set>` functions

Article • 12/07/2021

swap
swap (hash_multiset)

swap

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Exchanges the elements of two hash_sets.

```
C++
```

```
void swap(
    hash_set <Key, Traits, Allocator>& left,
    hash_set <Key, Traits, Allocator>& right);
```

Parameters

right
The hash_set providing the elements to be swapped, or the hash_set whose elements are to be exchanged with those of the hash_set *left*.

left
The hash_set whose elements are to be exchanged with those of the hash_set *right*.

Remarks

The `swap` template function is an algorithm specialized on the container class hash_set to execute the member function `left.`swap(`right`). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class hash_set::swap for an example that uses the template version of `swap`.

swap (hash_multiset)

> ⓘ **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Exchanges the elements of two hash_multisets.

```
C++

void swap(hash_multiset <Key, Traits, Allocator>& left, hash_multiset <Key,
Traits, Allocator>& right);
```

Parameters

right
The hash_multiset providing the elements to be swapped, or the hash_multiset whose elements are to be exchanged with those of the hash_multiset *left*.

left
The hash_multiset whose elements are to be exchanged with those of the hash_multiset *right*.

Remarks

The `swap` template function is an algorithm specialized on the container class hash_multiset to execute the member function `left.swap(right)`. This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

template <class T> void swap(T&, T&),

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class hash_multiset::swap for an example that uses the template version of `swap`.

See also

<hash_set>

`<hash_set>` operators

Article • 12/07/2021

operator!=
operator!= (hash_multiset)
operator==
operator== (hash_multiset)

operator!=

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Tests if the hash_set object on the left side of the operator is not equal to the hash_set object on the right side.

```
C++
```

```
bool operator!=(const hash_set <Key, Traits, Allocator>& left, const
hash_set <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_set`.

right
An object of type `hash_set`.

Return Value

`true` if the hash_sets are not equal; **`false`** if hash_sets are equal.

Remarks

The comparison between hash_set objects is based on a pairwise comparison between their elements. Two hash_sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Members of the <hash_map> and <hash_set> header files are in the stdext Namespace.

Example

C++

```
// hash_set_op_ne.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i );
   }

   if ( hs1 != hs2 )
      cout << "The hash_sets hs1 and hs2 are not equal." << endl;
   else
      cout << "The hash_sets hs1 and hs2 are equal." << endl;

   if ( hs1 != hs3 )
      cout << "The hash_sets hs1 and hs3 are not equal." << endl;
   else
      cout << "The hash_sets hs1 and hs3 are equal." << endl;
}
```

Output

```
The hash_sets hs1 and hs2 are not equal.
The hash_sets hs1 and hs3 are equal.
```

operator==

> **Note**
>
> This API is obsolete. The alternative is **unordered_set Class**.

Tests if the hash_set object on the left side of the operator is equal to the hash_set object on the right side.

```
C++
```

```
bool operator!==(const hash_set <Key, Traits, Allocator>& left, const
hash_set <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_set`.

right
An object of type `hash_set`.

Return Value

`true` if the hash_set on the left side of the operator is equal to the hash_set on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between hash_set objects is based on a pairwise comparison of their elements. Two hash_sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// hash_set_op_eq.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
```

```
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The hash_sets s1 and s2 are equal." << endl;
   else
      cout << "The hash_sets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The hash_sets s1 and s3 are equal." << endl;
   else
      cout << "The hash_sets s1 and s3 are not equal." << endl;
}
```

Output

```
The hash_sets s1 and s2 are not equal.
The hash_sets s1 and s3 are equal.
```

operator!= (hash_multiset)

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Tests if the hash_multiset object on the left side of the operator is not equal to the hash_multiset object on the right side.

C++

```
bool operator!=(const hash_multiset <Key, Traits, Allocator>& left, const
hash_multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_multiset`.

right
An object of type `hash_multiset`.

Return Value

`true` if the hash_multisets are not equal; `false` if hash_multisets are equal.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison between their elements. Two hash_multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// hashset_op_ne.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> hs1, hs2, hs3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      hs1.insert ( i );
      hs2.insert ( i * i );
      hs3.insert ( i );
   }

   if ( hs1 != hs2 )
      cout << "The hash_multisets hs1 and hs2 are not equal." << endl;
   else
      cout << "The hash_multisets hs1 and hs2 are equal." << endl;

   if ( hs1 != hs3 )
      cout << "The hash_multisets hs1 and hs3 are not equal." << endl;
   else
      cout << "The hash_multisets hs1 and hs3 are equal." << endl;
}
```

Output

```
The hash_multisets hs1 and hs2 are not equal.
The hash_multisets hs1 and hs3 are equal.
```

operator== (hash_multiset)

> ⓘ Note
>
> This API is obsolete. The alternative is **unordered_set Class**.

Tests if the hash_multiset object on the left side of the operator is equal to the hash_multiset object on the right side.

C++

```
bool operator!==(const hash_multiset <Key, Traits, Allocator>& left, const
hash_multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `hash_multiset`.

right
An object of type `hash_multiset`.

Return Value

`true` if the hash_multiset on the left side of the operator is equal to the hash_multiset on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between hash_multiset objects is based on a pairwise comparison of their elements. Two hash_multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// hash_multiset_op_eq.cpp
// compile with: /EHsc
#include <hash_set>
#include <iostream>

int main( )
{
   using namespace std;
   using namespace stdext;
   hash_multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The hash_multisets s1 and s2 are equal." << endl;
   else
      cout << "The hash_multisets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The hash_multisets s1 and s2 are equal." << endl;
   else
      cout << "The hash_multisets s1 and s2 are not equal." << endl;
}
```

Output

```
The hash_multisets s1 and s2 are not equal.
The hash_multisets s1 and s2 are equal.
```

See also

<hash_set>

<initializer_list>

Article • 12/07/2021

Defines the container class template `initializer_list` and several supporting templates.

Syntax

```
C++

#include <initializer_list>
```

Classes

Class	Description
initializer_list	Provides access to an array of elements in which each member is of the specified type.

See also

Header Files Reference

initializer_list class

Article • 08/03/2021

Provides access to an array of elements in which each member is of the specified type.

Syntax

```
C++

template <class Type>
class initializer_list
```

Parameters

Type
The element data type to be stored in the `initializer_list`.

Remarks

An `initializer_list` can be constructed using a braced initializer list:

```
C++

initializer_list<int> i1{ 1, 2, 3, 4 };
```

The compiler transforms braced initializer lists with homogeneous elements into an `initializer_list` whenever the function signature requires an `initializer_list`. For more information about using `initializer_list`, see Uniform initialization and delegating constructors

Constructors

Constructor	Description
initializer_list	Constructs an object of type `initializer_list`.

Typedefs

Type name	Description
`value_type`	The type of the elements in the `initializer_list`.
`reference`	A type that provides a reference to an element in the `initializer_list`.
`const_reference`	A type that provides a constant reference to an element in the `initializer_list`.
`size_type`	A type that represents the number of elements in the `initializer_list`.
`iterator`	A type that provides an iterator for the `initializer_list`.
`const_iterator`	A type that provides a constant iterator for the `initializer_list`.

Member functions

Member function	Description
begin	Returns a pointer to the first element in an `initializer_list`.
end	Returns a pointer to one past the last element in an `initializer_list`.
size	Returns the number of elements in the `initializer_list`.

Requirements

Header: <initializer_list>

Namespace: std

initializer_list::begin

Returns a pointer to the first element in an `initializer_list`.

```
C++
```

```
constexpr const InputIterator* begin() const noexcept;
```

Return value

A pointer to the first element of the `initializer_list`. If the list is empty, the pointer is the same for the beginning and end of the list.

initializer_list::end

Returns a pointer to one past the last element in an `initializer list`.

```
C++
```

```
constexpr const InputIterator* end() const noexcept;
```

Return value

A pointer to one past the last element in the list. If the list is empty, it's the same as the pointer to the first element in the list.

initializer_list::initializer_list

Constructs an object of type `initializer_list`.

```
C++
```

```
constexpr initializer_list() noexcept;
initializer_list(const InputIterator First, const InputIterator Last);
```

Parameters

First
The position of the first element in the range of elements to be copied.

Last
The position of the first element beyond the range of elements to be copied.

Remarks

An `initializer_list` is based on an array of objects of the specified type. Copying an `initializer_list` creates a second instance of a list pointing to the same objects; the underlying objects aren't copied.

Example

```
C++
```

```cpp
// initializer_list_class.cpp
// compile with: /EHsc
#include <initializer_list>
#include <iostream>

int main()
{
    using namespace std;
    // Create an empty initializer_list c0
    initializer_list <int> c0;

    // Create an initializer_list c1 with 1 element
    initializer_list <int> c1{ 3 };

    // Create an initializer_list c2 with 5 elements
    initializer_list <int> c2{ 5, 4, 3, 2, 1 };

    // Create a copy, initializer_list c3, of initializer_list c2
    initializer_list <int> c3(c2);

    // Create a initializer_list c4 by copying the range c3[ first,  last)
    const int* c3_ptr = c3.begin();
    c3_ptr++;
    c3_ptr++;
    initializer_list <int> c4(c3.begin(), c3_ptr);

    // Move initializer_list c4 to initializer_list c5
    initializer_list <int> c5(move(c4));

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    cout << "c2 =";
    for (auto c : c2)
        cout << " " << c;
    cout << endl;

    cout << "c3 =";
    for (auto c : c3)
        cout << " " << c;
    cout << endl;

    cout << "c5 =";
    for (auto c : c5)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 3
c2 = 5 4 3 2 1
c3 = 5 4 3 2 1
c5 = 5 4
```

initializer_list::size

Returns the number of elements in the list.

C++

```
constexpr size_t size() const noexcept;
```

Return value

The number of elements in the list.

See also

<forward_list>

`<iomanip>`

Article • 03/02/2022

Include the `iostreams` standard header `<iomanip>` to define several manipulators that each take a single argument.

Syntax

```
C++

#include <iomanip>
```

Remarks

Each of these manipulators returns an unspecified type, called `T1` through `T10`, that overloads both `basic_istream<Elem, Tr>::operator>>` and `basic_ostream<Elem, Tr>::operator<<`. For more information, see operator>> and operator<<.

Manipulators

Name	Description
get_money	Obtains a monetary amount, optionally in international format.
get_time	Obtains a time in a time structure by using a specified format.
put_money	Provides a monetary amount, optionally in international format.
put_time	Provides a time in a time structure and a format string to use.
quoted	Enables convenient round-tripping of strings with insertion and extraction operators.
resetiosflags	Clears the specified flags.
setbase	Set base for integers.
setfill	Sets the character that will be used to fill spaces in a right-justified display.
setiosflags	Sets the specified flags.
setprecision	Sets the precision for floating-point values.

Name	Description
setw	Specifies the width of the display field.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<iomanip>` functions

Article • 11/22/2021

get_money
get_time
put_money
put_time
quoted
resetiosflags
setbase
setfill
setiosflags
setprecision
setw

`get_money`

Extracts a monetary value from a stream using the specified format, and returns the value in a parameter.

```
C++
```

```
template <class Money>
T7 get_money(Money& amount, bool use_intl);
```

Parameters

`amount`
The extracted monetary value.

`use_intl`
If **`true`**, use international format. The default value is **`false`**.

Remarks

The manipulator returns an object that, when extracted from the stream `str`, behaves as a `formatted input function` that calls the member function `get` for the locale facet `money_get` associated with `str`, using *`use_intl`* to indicate international format. If

successful, the call stores in *amount* the extracted monetary value. The manipulator then returns `str`.

`Money` must be of type **`long double`** or an instantiation of `basic_string` with the same element and traits parameters as `str`.

`get_time`

Extracts a time value from a stream using the specified format. Returns the value in a parameter as a time structure.

C++

```
template <class Elem>
T10 get_time(struct tm *time_ptr, const Elem *time_format);
```

Parameters

time_ptr

The time in the form of a time structure.

time_format

The format to use to get the time value.

Remarks

The manipulator returns an object that, when extracted from the stream `str`, behaves as a `formatted input function` that calls the member function `get` for the locale facet `time_get` associated with `str`, using `tptr` to indicate the time structure and `fmt` to indicate the beginning of a null-terminated format string. If successful, the call stores in the time structure the values associated with any extracted time fields. The manipulator then returns `str`.

Example

C++

```
#include <ctime>
#include <iomanip>
#include <iostream>

int main()
```

```
{
    std::cout << "Enter a time, for example 15:24 for 3:24pm: ";
    struct std::tm when;
    std::cin >> std::get_time(&when, "%R");
    if (!std::cin.fail())
    {
        std::cout << "Entered: " << when.tm_hour << " hours, " <<
when.tm_min << " minutes\n";
    }

    return (int)std::cin.fail();
}
```

put_money

Inserts a monetary amount using the specified format into a stream.

C++

```
template <class Money>
T8 put_money(const Money& amount, bool use_intl);
```

Parameters

amount

The monetary amount to insert into the stream.

use_intl

Set to **true** if manipulator should use international format, **false** if it shouldn't.

Return Value

Returns `str`.

Remarks

The manipulator returns an object that, when inserted into the stream `str`, behaves as a formatted output function that calls the member function `put` for the locale facet `money_put` associated with `str`. If successful, the call inserts `amount` suitably formatted, using *use_intl* to indicate international format and `str.fill()`, as the fill element. The manipulator then returns `str`.

`Money` must be of type **`long double`** or an instantiation of `basic_string` with the same element and traits parameters as `str`.

`put_time`

Writes a time value from a time structure to a stream by using a specified format.

C++

```
template <class Elem>
T10 put_time(struct tm* time_ptr, const Elem* time_format);
```

Parameters

`time_ptr`
The time value to write to the stream, provided in a time structure.

`time_format`
The format to write the time value.

Remarks

The manipulator returns an object that, when inserted into the stream `str`, behaves as a `formatted output function`. The output function calls the member function `put` for the locale facet `time_put` associated with `str`. The output function uses *`time_ptr`* to indicate the time structure and *`time_format`* to indicate the beginning of a null-terminated format string. If successful, the call inserts literal text from the format string and converted values from the time structure. The manipulator then returns `str`.

`quoted`

(New in C++14) An `iostream` manipulator that enables convenient round-tripping of strings into and out of streams using the `>>` and `<<` operators.

C++

```
quoted(std::string str) // or wstring
quoted(const char* str) //or wchar_t*
quoted(std::string str, char delimiter, char escape) // or wide versions
quoted(const char* str, char delimiter, char escape) // or wide versions
```

Parameters

str

A `std::string`, `char*`, string literal or raw string literal, or a wide version of any of these (for example, `std::wstring`, `wchar_t*`).

delimiter

A user-specified character, or wide character, to use as the delimiter for the beginning and end of the string.

escape

A user-specified character, or wide character, to use as the escape character for escape sequences within the string.

Remarks

See Using Insertion Operators and Controlling Format.

Examples

This example shows how to use `quoted` with the default delimiter and escape character using narrow strings. Wide strings are equally supported.

C++

```
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

void show_quoted_v_nonquoted()
{
    // Results are identical regardless of input string type:
    // string inserted { R"(This is a "sentence".)" }; // raw string literal
    // string inserted { "This is a \"sentence\"." };  // regular string
literal
    const char* inserted = "This is a \"sentence\".";  // const char*
    stringstream ss, ss_quoted;
    string extracted, extracted_quoted;

    ss << inserted;
    ss_quoted << quoted(inserted);

    cout << "ss.str() is storing       : " << ss.str() << endl;
    cout << "ss_quoted.str() is storing: " << ss_quoted.str() << endl <<
endl;
```

```
    // Round-trip the strings
    ss >> extracted;
    ss_quoted >> quoted(extracted_quoted);

    cout << "After round trip: " << endl;
    cout << "Non-quoted      : " << extracted << endl;
    cout << "Quoted          : " << extracted_quoted << endl;
}

int main(int argc, char* argv[])
{
    show_quoted_v_nonquoted();

    // Keep console window open in debug mode.
    cout << endl << "Press Enter to exit" << endl;
    string input{};
    getline(cin, input);
}

/* Output:
ss.str() is storing       : This is a "sentence".
ss_quoted.str() is storing: "This is a \"sentence\"."

After round trip:
Non-quoted      : This
Quoted          : This is a "sentence".

Press Enter to exit
*/
```

The following example shows how to provide a custom delimiter or escape character:

C++

```
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

void show_custom_delimiter()
{
    string inserted{ R"("This" "is" "a" "heavily-quoted" "sentence".)" };
    // string inserted{ "\"This\" \"is\" \"a\" \"heavily-quoted\"
\"sentence\"" };
    // const char* inserted{ "\"This\" \"is\" \"a\" \"heavily-quoted\"
\"sentence\"" };
    stringstream ss, ss_quoted;
    string extracted;

    ss_quoted << quoted(inserted, '*');
    ss << inserted;
```

```
    cout << "ss_quoted.str() is storing: " << ss_quoted.str() << endl;
    cout << "ss.str() is storing       : " << ss.str() << endl << endl;

    // Use the same quoted arguments as on insertion.
    ss_quoted >> quoted(extracted, '*');

    cout << "After round trip: " << endl;
    cout << "Quoted          : " << extracted << endl;

    extracted = {};
    ss >> extracted;
    cout << "Non-quoted      : " << extracted << endl << endl;
}

void show_custom_escape()
{
    string inserted{ R"(\\root\trunk\branch\nest\egg\yolk)" };
    // string inserted{ "\\\\root\\trunk\\branch\\nest\\egg\\yolk" };
    stringstream ss, ss_quoted, ss_quoted_custom;
    string extracted;

    // Use '"' as delimiter and '~' as escape character.
    ss_quoted_custom << quoted(inserted, '"', '~');
    ss_quoted << quoted(inserted);
    ss << inserted;
    cout << "ss_quoted_custom.str(): " << ss_quoted_custom.str() << endl;
    cout << "ss_quoted.str()       : " << ss_quoted.str() << endl;
    cout << "ss.str()              : " << ss.str() << endl << endl;

    // No spaces in this string, so non-quoted behaves same as quoted
    // after round-tripping.
}

int main(int argc, char* argv[])
{
    cout << "Custom delimiter:" << endl;
    show_custom_delimiter();
    cout << "Custom escape character:" << endl;
    show_custom_escape();

    // Keep console window open in debug mode.
    cout << endl << "Press Enter to exit" << endl;
    string input{};
    getline(cin, input);
}
/* Output:
Custom delimiter:
ss_quoted.str() is storing: *"This" "is" "a" "heavily-quoted" "sentence".*
ss.str() is storing       : "This" "is" "a" "heavily-quoted" "sentence".

After round trip:
Quoted          : "This" "is" "a" "heavily-quoted" "sentence".
Non-quoted      : "This"

Custom escape character:
```

```
ss_quoted_custom.str(): "\\root\trunk\branch\nest\egg\yolk"
ss_quoted.str()       : "\\\\root\\trunk\\branch\\nest\\egg\\yolk"
ss.str()              : \\root\trunk\branch\nest\egg\yolk

Press Enter to exit
*/
```

`resetiosflags`

Clears the specified flags.

```
C++
```

```
T1 resetiosflags(ios_base::fmtflags mask);
```

Parameters

`mask`

The flags to clear.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(ios_base::fmtflags, mask)`, and then returns `str`, see setf and fmtflags.

Example

See setw for an example of using `resetiosflags`.

`setbase`

Set base for integers.

```
C++
```

```
T3 setbase(int base);
```

Parameters

base

The number base.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(mask, ios_base::basefield)`, and then returns `str`, see ios_base::basefield. Here, `mask` is determined as follows:

- If *base* is 8, then `mask` is ios_base::oct.
- If *base* is 10, then mask is ios_base::dec.
- If *base* is 16, then `mask` is ios_base::hex.
- If *base* is any other value, then mask is ios_base::fmtflags(0)`.

Example

See setw for an example of using `setbase`.

`setfill`

Sets the character that will be used to fill spaces in a right-justified display.

```
C++
```

```
template <class Elem>
T4 setfill(Elem Ch);
```

Parameters

Ch

The character that will be used to fill spaces in a right-justified display.

Return Value

The template manipulator returns an object that, when extracted from or inserted into the stream `str`, calls str.fill(Ch), and then returns `str`. The type `Elem` must be the same as the element type for the stream `str`.

Example

See setw for an example of using `setfill`.

`setiosflags`

Sets the specified flags.

```
C++

T2 setiosflags(ios_base::fmtflags mask);
```

Parameters

`mask`
The flags to set.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(mask)`, and then returns `str`, see setf.

Example

See setw for an example of using `setiosflags`.

`setprecision`

Sets the precision for floating-point values.

```
C++

T5 setprecision(streamsize Prec);
```

Parameters

`Prec`
The precision for floating-point values.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.precision(Prec)`, and then returns `str`, see precision.

Example

See setw for an example of using `setprecision`.

`setw`

Specifies the width of the display field for the next element in the stream.

```
C++
```

```
T6 setw(streamsize Wide);
```

Parameters

`Wide`
The width of the display field.

Return Value

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.width(Wide)`, then returns `str`. For more information, see width.

Remarks

`setw` sets the width only for the next element in the stream and must be inserted before each element whose width you want to specify.

Example

```
C++
```

```
// iomanip_setw.cpp
// compile with: /EHsc
// Defines the entry point for the console application.
//
// Sample use of the following manipulators:
```

```
//   resetiosflags
//   setiosflags
//   setbase
//   setfill
//   setprecision
//   setw

#include <iostream>
#include <iomanip>

using namespace std;

const double   d1 = 1.23456789;
const double   d2 = 12.3456789;
const double   d3 = 123.456789;
const double   d4 = 1234.56789;
const double   d5 = 12345.6789;
const long      l1 = 16;
const long      l2 = 256;
const long      l3 = 1024;
const long      l4 = 4096;
const long      l5 = 65536;
int         base = 10;

void DisplayDefault( )
{
   cout << endl << "default display" << endl;
   cout << "d1 = " << d1 << endl;
   cout << "d2 = " << d2 << endl;
   cout << "d3 = " << d3 << endl;
   cout << "d4 = " << d4 << endl;
   cout << "d5 = " << d5 << endl;
}

void DisplayWidth( int n )
{
   cout << endl << "fixed width display set to " << n << ".\n";
   cout << "d1 = " << setw(n) << d1 << endl;
   cout << "d2 = " << setw(n) << d2 << endl;
   cout << "d3 = " << setw(n) << d3 << endl;
   cout << "d4 = " << setw(n) << d4 << endl;
   cout << "d5 = " << setw(n) << d5 << endl;
}

void DisplayLongs( )
{
   cout << setbase(10);
   cout << endl << "setbase(" << base << ")" << endl;
   cout << setbase(base);
   cout << "l1 = " << l1 << endl;
   cout << "l2 = " << l2 << endl;
   cout << "l3 = " << l3 << endl;
   cout << "l4 = " << l4 << endl;
   cout << "l5 = " << l5 << endl;
}
```

```
int main( int argc, char* argv[] )
{
   DisplayDefault( );

   cout << endl << "setprecision(" << 3 << ")" << setprecision(3);
   DisplayDefault( );

   cout << endl << "setprecision(" << 12 << ")" << setprecision(12);
   DisplayDefault( );

   cout << setiosflags(ios_base::scientific);
   cout << endl << "setiosflags(" << ios_base::scientific << ")";
   DisplayDefault( );

   cout << resetiosflags(ios_base::scientific);
   cout << endl << "resetiosflags(" << ios_base::scientific << ")";
   DisplayDefault( );

   cout << endl << "setfill('" << 'S' << "')" << setfill('S');
   DisplayWidth(15);
   DisplayDefault( );

   cout << endl << "setfill('" << ' ' << "')" << setfill(' ');
   DisplayWidth(15);
   DisplayDefault( );

   cout << endl << "setprecision(" << 8 << ")" << setprecision(8);
   DisplayWidth(10);
   DisplayDefault( );

   base = 16;
   DisplayLongs( );

   base = 8;
   DisplayLongs( );

   base = 10;
   DisplayLongs( );

   return   0;
}
```

Output

```
default display
d1 = 1.23457
d2 = 12.3457
d3 = 123.457
d4 = 1234.57
d5 = 12345.7

setprecision(3)
```

```
default display
d1 = 1.23
d2 = 12.3
d3 = 123
d4 = 1.23e+003
d5 = 1.23e+004

setprecision(12)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setiosflags(4096)
default display
d1 = 1.234567890000e+000
d2 = 1.234567890000e+001
d3 = 1.234567890000e+002
d4 = 1.234567890000e+003
d5 = 1.234567890000e+004

resetiosflags(4096)
default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setfill('S')
fixed width display set to 15.
d1 = SSSSS1.23456789
d2 = SSSSS12.3456789
d3 = SSSSS123.456789
d4 = SSSSS1234.56789
d5 = SSSSS12345.6789

default display
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setfill(' ')
fixed width display set to 15.
d1 =      1.23456789
d2 =      12.3456789
d3 =      123.456789
d4 =      1234.56789
d5 =      12345.6789

default display
```

```
d1 = 1.23456789
d2 = 12.3456789
d3 = 123.456789
d4 = 1234.56789
d5 = 12345.6789

setprecision(8)
fixed width display set to 10.
d1 =  1.2345679
d2 =  12.345679
d3 =  123.45679
d4 =  1234.5679
d5 =  12345.679

default display
d1 = 1.2345679
d2 = 12.345679
d3 = 123.45679
d4 = 1234.5679
d5 = 12345.679

setbase(16)
l1 = 10
l2 = 100
l3 = 400
l4 = 1000
l5 = 10000

setbase(8)
l1 = 20
l2 = 400
l3 = 2000
l4 = 10000
l5 = 200000

setbase(10)
l1 = 16
l2 = 256
l3 = 1024
l4 = 4096
l5 = 65536
```

See also

<iomanip>

`<ios>`

Article • 12/07/2021

Defines several types and functions basic to the operation of iostreams. This header is typically included for you by another iostream headers; you rarely include it directly.

Requirements

Header: <ios>

Namespace: std

> **Note**
>
> The <ios> library uses the `#include <iosfwd>` statement.

Remarks

A large group of functions are manipulators. A manipulator declared in <ios> alters the values stored in its argument object of class ios_base. Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the class templates basic_istream or basic_ostream. For example, noskipws(**str**) clears the format flag `ios_base::skipws` in the object `str`, which can be of one of these types.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, because of special insertion and extraction operations supplied for the classes derived from `ios_base`. For example:

```
C++

istr>> noskipws;
```

calls noskipws(**istr**).

Members

Typedefs

Name	Description
ios	Supports the ios class from the old iostream library.
streamoff	Supports internal operations.
streampos	Holds the current position of the buffer pointer or file pointer.
streamsize	Specifies the size of the stream.
wios	Supports the wios class from the old iostream library.
wstreampos	Holds the current position of the buffer pointer or file pointer.

Manipulators

Name	Description
boolalpha	Specifies that variables of type bool appear as `true` or `false` in the stream.
dec	Specifies that integer variables appear in base 10 notation.
defaultfloat	Configures the flags of an `ios_base` object to use a default display format for float values.
fixed	Specifies that a floating-point number is displayed in fixed-decimal notation.
hex	Specifies that integer variables appear in base 16 notation.
hexfloat	
internal	Causes a number's sign to be left justified and the number to be right justified.
left	Causes text that is not as wide as the output width to appear in the stream flush with the left margin.
noboolalpha	Specifies that variables of type bool appear as 1 or 0 in the stream.
noshowbase	Turns off indicating the notational base in which a number is displayed.
noshowpoint	Displays only the whole-number part of floating-point numbers whose fractional part is zero.
noshowpos	Causes positive numbers to not be explicitly signed.
noskipws	Cause spaces to be read by the input stream.
nounitbuf	Causes output to be buffered and processed when the buffer is full.
nouppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.

Name	Description
oct	Specifies that integer variables appear in base 8 notation.
right	Causes text that is not as wide as the output width to appear in the stream flush with the right margin.
scientific	Causes floating point numbers to be displayed using scientific notation.
showbase	Indicates the notational base in which a number is displayed.
showpoint	Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.
showpos	Causes positive numbers to be explicitly signed.
skipws	Cause spaces to not be read by the input stream.
unitbuf	Causes output to be processed when the buffer is not empty.
uppercase	Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

Error Reporting

Name	Description
io_errc	
is_error_code_enum	
iostream_category	
make_error_code	
make_error_condition	

Classes

Name	Description
basic_ios	The class template describes the storage and member functions common to both input streams (of class template basic_istream) and output streams (of class template basic_ostream) that depend on the template parameters.
fpos	The class template describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream.

Name	Description
ios_base	The class describes the storage and member functions common to both input and output streams that do not depend on the template parameters.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<ios>` functions

Article • 08/01/2022

boolalpha

Specifies that variables of type bool appear as `true` or `false` in the stream.

C++

```
ios_base& boolalpha(ios_base& str);
```

Parameters

str

A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, variables of type `bool` are displayed as 1 or 0.

`boolalpha` effectively calls `str.`setf(`ios_base::boolalpha`), and then returns *str*.

noboolalpha reverses the effect of `boolalpha`.

Example

C++

```
// ios_boolalpha.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   bool b = true;
   cout << b << endl;
```

```
    boolalpha( cout );
    cout << b << endl;
    noboolalpha( cout );
    cout << b << endl;
    cout << boolalpha << b << endl;
}
```

Output

```
1
true
1
true
```

dec

Specifies that integer variables appear in base 10 notation.

C++

```
ios_base& dec(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, integer variables are displayed in base 10.

`dec` effectively calls `str.`setf(`ios_base::dec`, `ios_base::basefield`), and then returns *str*.

Example

C++

```cpp
// ios_dec.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 100;

   cout << i << endl;   // Default is base 10
   cout << hex << i << endl;
   dec( cout );
   cout << i << endl;
   oct( cout );
   cout << i << endl;
   cout << dec << i << endl;
}
```

Output

```
100
64
100
144
100
```

defaultfloat

Configures the flags of an `ios_base` object to use a default display format for float values.

C++

```cpp
ios_base& defaultfloat(ios_base& iosbase);
```

Parameters

_Iosbase
An `ios_base` object.

Remarks

The manipulator effectively calls `iosbase.`ios_base::unsetf`(ios_base::floatfield)`, then returns *iosbase*.

fixed

Specifies that a floating-point number is displayed in fixed-decimal notation.

C++

```
ios_base& fixed(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

`fixed` is the default display notation for floating-point numbers. scientific causes floating-point numbers to be displayed using scientific notation.

The manipulator effectively calls *str*.setf(`ios_base::fixed`, `ios_base::floatfield`), and then returns *str*.

Example

C++

```
// ios_fixed.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   float i = 1.1F;

   cout << i << endl;   // fixed is the default
   cout << scientific << i << endl;
   cout.precision( 1 );
   cout << fixed << i << endl;
}
```

Output

```
1.1
1.100000e+000
1.1
```

hex

Specifies that integer variables shall appear in base 16 notation.

C++

```
ios_base& hex(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, integer variables are displayed in base 10 notation. dec and oct also change the way integer variables appear.

The manipulator effectively calls `str`.setf(`ios_base::hex`, `ios_base::basefield`), and then returns *str*.

Example

See dec for an example of how to use `hex`.

hexfloat

C++

```
ios_base& hexfloat (ios_base& str);
```

io_errc

```
C++
```

```
enum class io_errc {
    stream = 1
};
```

internal

Causes a number's sign to be left justified and the number to be right justified.

```
C++
```

```
ios_base& internal(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

showpos causes the sign to display for positive numbers.

The manipulator effectively calls `str.setf(ios_base::internal, ios_base::adjustfield)`, and then returns *str*.

Example

```
C++
```

```
// ios_internal.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>

int main( void )
```

```
{
   using namespace std;
   float i = -123.456F;
   cout.fill( '.' );
   cout << setw( 10 ) << i << endl;
   cout << setw( 10 ) << internal << i << endl;
}
```

Output

```
..-123.456
-..123.456
```

is_error_code_enum

C++

```
template <> struct is_error_code_enum<io_errc> : public true_type { };
```

iostream_category

C++

```
const error_category& iostream_category() noexcept;
```

left

Causes text that is not as wide as the output width to appear in the stream flush with the left margin.

C++

```
ios_base& left(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

The manipulator effectively calls `str.setf(ios_base::left, ios_base::adjustfield)`, and then returns *str*.

Example

C++

```
// ios_left.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   double f1= 5.00;
   cout.width( 20 );
   cout << f1 << endl;
   cout << left << f1 << endl;
}
```

Output

```
5
        5
```

make_error_code

C++

```
error_code make_error_code(io_errc e) noexcept;
```

make_error_condition

C++

```
error_condition make_error_condition(io_errc e) noexcept;
```

noboolalpha

Specifies that variables of type bool appear as 1 or 0 in the stream.

```
C++
```

```
ios_base& noboolalpha(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, `noboolalpha` is in effect.

`noboolalpha` effectively calls `str.`unsetf`(ios_base::boolalpha)`, and then returns *str*.

boolalpha reverses the effect of `noboolalpha`.

Example

See boolalpha for an example of using `noboolalpha`.

noshowbase

Turns off indicating the notational base in which a number is displayed.

```
C++
```

```
ios_base& noshowbase(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

`noshowbase` is on by default. Use showbase to indicate the notational base of numbers.

The manipulator effectively calls `str.`unsetf`(ios_base::showbase)`, and then returns *str*.

Example

See showbase for an example of how to use `noshowbase`.

noshowpoint

Displays only the whole-number part of floating-point numbers whose fractional part is zero.

```
C++
```

```
ios_base& noshowpoint(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

`noshowpoint` is on by default; use showpoint and precision to display zeros after the decimal point.

The manipulator effectively calls `str.unsetf(ios_base::showpoint)`, and then returns *str*.

Example

C++

```
// ios_noshowpoint.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   double f1= 5.000;
   cout << f1 << endl;   // noshowpoint is default
   cout.precision( 4 );
   cout << showpoint << f1 << endl;
   cout << noshowpoint << f1 << endl;
}
```

Output

```
5
5.000
5
```

noshowpos

Causes positive numbers to not be explicitly signed.

C++

```
ios_base& noshowpos(ios_base& str);
```

Parameters

str

A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

`noshowpos` is on by default.

The manipulator effectively calls `str.`unsetf`(ios_base::showpos)`, then returns *str*.

Example

See showpos for an example of using `noshowpos`.

noskipws

Cause spaces to be read by the input stream.

```
C++
```

```
ios_base& noskipws(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, skipws is in effect. When a space is read in the input stream, it signals the end of the buffer.

The manipulator effectively calls `str.`unsetf`(ios_base::skipws)`, and then returns *str*.

Example

```
C++
```

```
// ios_noskipws.cpp
// compile with: /EHsc
#include <iostream>
```

```
#include <string>

int main() {
   using namespace std;
   string s1, s2, s3;
   cout << "Enter three strings: ";
   cin >> noskipws >> s1 >> s2 >> s3;
   cout << "." << s1  << "." << endl;
   cout << "." << s2 << "." << endl;
   cout << "." << s3 << "." << endl;
}
```

nounitbuf

Causes output to be buffered and processed on when the buffer is full.

C++

```
ios_base& nounitbuf(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

unitbuf causes the buffer to be processed when it is not empty.

The manipulator effectively calls `str.`unsetf`(ios_base::unitbuf)`, and then returns *str*.

nouppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in lowercase.

C++

```
ios_base& nouppercase(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

The manipulator effectively calls `str.`unsetf`(ios_base::uppercase)`, and then returns *str*.

Example

See uppercase for an example of using `nouppercase`.

oct

Specifies that integer variables appear in base 8 notation.

```
C++

ios_base& oct(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, integer variables are displayed in base 10 notation. dec and hex also change the way integer variables appear.

The manipulator effectively calls `str.setf(ios_base::oct, ios_base::basefield)`, and then returns *str*.

Example

See dec for an example of how to use `oct`.

right

Causes text that is not as wide as the output width to appear in the stream flush with the right margin.

```
C++
```

```
ios_base& right(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

left also modifies the justification of text.

The manipulator effectively calls `str.setf(ios_base::right, ios_base::adjustfield)`, and then returns *str*.

Example

```
C++
```

```
// ios_right.cpp
// compile with: /EHsc
#include <iostream>

int main( )
```

```
{
   using namespace std;
   double f1= 5.00;
   cout << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
   cout.width( 20 );
   cout << left << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
   cout.width( 20 );
   cout << right << f1 << endl;
   cout.width( 20 );
   cout << f1 << endl;
}
```

Output

```
5
                    5
5
5
                    5
                    5
```

scientific

Causes floating-point numbers to be displayed using scientific notation.

C++

```
ios_base& scientific(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, fixed notation is in effect for floating-point numbers.

The manipulator effectively calls `str.setf(ios_base::scientific, ios_base::floatfield)`, and then returns *str*.

Example

C++

```
// ios_scientific.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   float i = 100.23F;

   cout << i << endl;
   cout << scientific << i << endl;
}
```

Output

```
100.23
1.002300e+002
```

showbase

Indicates the notational base in which a number is displayed.

C++

```
ios_base& showbase(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

The notational base of a number can be changed with dec, oct, or hex.

The manipulator effectively calls `str.setf(ios_base::showbase)`, and then returns *str*.

Example

C++

```
// ios_showbase.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int j = 100;

   cout << showbase << j << endl;   // dec is default
   cout << hex << j << showbase << endl;
   cout << oct << j << showbase << endl;

   cout << dec << j << noshowbase << endl;
   cout << hex << j << noshowbase << endl;
   cout << oct << j << noshowbase << endl;
}
```

Output

```
100
0x64
0144
100
64
144
```

showpoint

Displays the whole-number part of a floating-point number and digits to the right of the decimal point even when the fractional part is zero.

C++

```
ios_base& showpoint(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, noshowpoint is in effect.

The manipulator effectively calls `str.setf(ios_base::showpoint)`, and then returns *str*.

Example

See noshowpoint for an example of using `showpoint`.

showpos

Causes positive numbers to be explicitly signed.

```
C++
```

```
ios_base& showpos(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

noshowpos is the default.

The manipulator effectively calls `str.setf(ios_base::showpos)`, and then returns *str*.

Example

C++

```
// ios_showpos.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   int i = 1;

   cout << noshowpos << i << endl;   // noshowpos is default
   cout << showpos << i << endl;
}
```

Output

```
1
+1
```

skipws

Cause spaces to not be read by the input stream.

C++

```
ios_base& skipws(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, `skipws` is in effect. noskipws will cause spaces to be read from the input stream.

The manipulator effectively calls `str.setf(ios_base::skipws)`, and then returns *str*.

Example

C++

```
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   char s1, s2, s3;
   cout << "Enter three characters: ";
   cin >> skipws >> s1 >> s2 >> s3;
   cout << "." << s1  << "." << endl;
   cout << "." << s2 << "." << endl;
   cout << "." << s3 << "." << endl;
}
```

Input

```
1 2 3
```

Output

```
Enter three characters: 1 2 3
.1.
.2.
.3.
```

unitbuf

Causes output to be processed when the buffer is not empty.

C++

```
ios_base& unitbuf(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

Note that `endl` also flushes the buffer.

nounitbuf is in effect by default.

The manipulator effectively calls `str.setf(ios_base::unitbuf)`, and then returns *str*.

uppercase

Specifies that hexadecimal digits and the exponent in scientific notation appear in uppercase.

```
C++
```

```
ios_base& uppercase(ios_base& str);
```

Parameters

str
A reference to an object of type ios_base, or to a type that inherits from `ios_base`.

Return Value

A reference to the object from which *str* is derived.

Remarks

By default, nouppercase is in effect.

The manipulator effectively calls `str.setf(ios_base::uppercase)`, and then returns *str*.

Example

C++

```cpp
// ios_uppercase.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
   using namespace std;

   double i = 1.23e100;
   cout << i << endl;
   cout << uppercase << i << endl;

   int j = 10;
   cout << hex << nouppercase << j << endl;
   cout << hex << uppercase << j << endl;
}
```

Output

```
1.23e+100
1.23E+100
a
A
```

`<ios>` typedefs

Article • 08/03/2021

`ios`

Supports the `ios` class from the old `iostream` library.

```
C++
```

```
typedef basic_ios<char, char_traits<char>> ios;
```

Remarks

The type is a synonym for class template basic_ios, specialized for elements of type `char` with default character traits.

`streamoff`

Supports internal operations.

```
C++
```

```
#ifdef _WIN64
    typedef __int64 streamoff;
#else
    typedef long streamoff;
#endif
```

Remarks

The type is a signed integer. It describes an object that can store a byte offset in stream positioning operations. Its representation has at least 32 value bits. It isn't necessarily large enough to represent an arbitrary byte position within a stream. The value `streamoff(-1)` generally indicates an erroneous offset.

`streampos`

Holds the current position of the buffer pointer or file pointer.

C++

```
typedef fpos<mbstate_t> streampos;
```

Remarks

The type is a synonym for fpos< `mbstate_t` >.

Example

C++

```
// ios_streampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   ofstream x( "iostream.txt" );
   x << "testing";
   streampos y = x.tellp( );
   cout << streamoff( y ) << '\n';
}
```

Output

```
7
```

`streamsize`

Denotes the size of the stream.

C++

```
#ifdef _WIN64
    typedef __int64 streamsize;
#else
    typedef int streamsize;
#endif
```

Remarks

The type is a signed integer that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It isn't necessarily large enough to represent an arbitrary byte position within a stream.

Example

After compiling and running the following program, look at the file `test.txt` to see the effect of setting `streamsize`.

C++

```
// ios_streamsize.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   char a[16] = "any such text";
   ofstream x( "test.txt" );
   streamsize y = 6;
   x.write( a, y );
}
```

`wios`

Supports the `wios` class from the old `iostream` library.

C++

```
typedef basic_ios<wchar_t, char_traits<wchar_t>> wios;
```

Remarks

The type is a synonym for class template basic_ios, specialized for elements of type `wchar_t` with default character traits.

`wstreampos`

Holds the current position of the buffer pointer or file pointer.

C++

```
typedef fpos<mbstate_t> wstreampos;
```

Remarks

The type is a synonym for fpos< `mbstate_t` >.

Example

C++

```
// ios_wstreampos.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   wofstream xw( "wiostream.txt" );
   xw << L"testing";
   wstreampos y = xw.tellp( );
   cout << streamoff( y ) << '\n';
}
```

Output

```
7
```

`basic_ios` Class

Article • 11/07/2022

The class template describes the storage and member functions common to both input streams (of class template basic_istream) and output streams (of class template basic_ostream) that depend on the template parameters. (The class ios_base describes what is common and not dependent on template parameters.) An object of class `basic_ios<class Elem, class Traits>` helps control a stream with elements of type `Elem`, whose character traits are determined by the class `Traits`.

Syntax

```
C++

template <class Elem, class Traits>
class basic_ios : public ios_base
```

Parameters

`Elem`

A character type.

`Traits`

A type providing information about the character type, defaults to `char_traits < Elem >`.

Remarks

An object of class `basic_ios<class Elem, class Traits>` stores:

- A tie pointer to an object of type `basic_istream<Elem, Traits>`.
- A stream buffer pointer to an object of type `basic_streambuf<Elem, Traits >`.
- Formatting information.
- Stream state information in a base object of type ios_base.
- A fill character in an object of type `char_type`.

For more information, see basic_istream and basic_streambuf.

Constructors

Constructor	Description
basic_ios	Constructs the `basic_ios` class.

Typedefs

Type name	Description
char_type	A synonym for the template parameter `Elem`.
int_type	A synonym for `Traits::int_type`.
off_type	A synonym for `Traits::off_type`.
pos_type	A synonym for `Traits::pos_type`.
traits_type	A synonym for the template parameter `Traits`.

Member functions

Member function	Description
bad	Indicates a loss of integrity of the stream buffer.
clear	Clears all error flags.
copyfmt	Copies flags from one stream to another.
eof	Indicates if the end of a stream has been reached.
exceptions	Indicates which exceptions will be thrown by the stream.
fail	Indicates failure to extract a valid field from a stream.
fill	Specifies or returns the character that will be used when the text isn't as wide as the stream.
good	Indicates the stream is in good condition.
imbue	Changes the locale.
init	Called by `basic_ios` constructors.

Member function	Description
move	Moves all values, except the pointer to the stream buffer, from the parameter to the current object.
narrow	Finds the equivalent char to a given `char_type`.
rdbuf	Routes stream to specified buffer.
rdstate	Reads the state of bits for flags.
set_rdbuf	Assigns a stream buffer to be the read buffer for this stream object.
setstate	Sets additional flags.
swap	Exchanges the values in this `basic_ios` object for those of another `basic_ios` object. The pointers to the stream buffers aren't swapped.
tie	Ensures that one stream is processed before another stream.
widen	Finds the equivalent `char_type` to a given char.

Operators

Operator	Description
explicit operator bool	Allows use of a `basic_ios` object as a **`bool`**. Automatic type conversion is disabled to prevent common, unintended side effects.
operator void *	Indicates if the stream is still good.
operator!	Indicates if the stream isn't bad.

Requirements

Header: `<ios>`

Namespace: `std`

`basic_ios::bad`

Indicates a loss of integrity of the stream buffer

```
```

C++

```
bool bad() const;
```

Return Value

`true` if `rdstate & badbit` is nonzero; otherwise **`false`**.

For more information on `badbit`, see ios_base::iostate.

Example

C++

```
// basic_ios_bad.cpp
// compile with: /EHsc
#include <iostream>

int main( void )
{
    using namespace std;
    bool b = cout.bad( );
    cout << boolalpha;
    cout << b << endl;

    b = cout.good( );
    cout << b << endl;
}
```

`basic_ios::basic_ios`

Constructs the `basic_ios` class.

C++

```
explicit basic_ios(basic_streambuf<Elem,  Traits>* sb);
basic_ios();
```

Parameters

`sb`

Standard buffer to store input or output elements.

Remarks

The first constructor initializes its member objects by calling `init(_ Sb)`. The second (protected) constructor leaves its member objects uninitialized. A later call to init must initialize the object before it can be safely destroyed.

`basic_ios::char_type`

A synonym for the template parameter `Elem`.

```
C++
```

```
typedef Elem char_type;
```

`basic_ios::clear`

Clears all error flags.

```
C++
```

```
void clear(iostate state = goodbit, bool reraise = false);
void clear(io_state state);
```

Parameters

`state`
(Optional) The flags you want to set after clearing all flags. Defaults to `goodbit`.

`reraise`
(Optional) Specifies whether the exception should be re-raised. Defaults to **`false`** (won't re-raise the exception).

Remarks

The flags are `goodbit`, `failbit`, `eofbit`, and `badbit`. Test for these flags with good, bad, eof, and fail

The member function replaces the stored stream state information with:

`state | (rdbuf != 0 goodbit : badbit)`

If `state&exceptions` is nonzero, it then throws an object of class failure.

For more information, see rdbuf and exceptions.

Example

See rdstate and getline for examples using `clear`.

`basic_ios::copyfmt`

Copies flags from one stream to another.

```
C++
```

```
basic_ios<Elem, Traits>& copyfmt(
const basic_ios<Elem, Traits>& right);
```

Parameters

`right`
The stream whose flags you want to copy.

Return Value

The **`this`** object for the stream to which you're copying the flags.

Remarks

The member function reports the callback event **`erase_event`**. It then copies from *`right`* into **`*this`** the fill character, the tie pointer, and the formatting information. Before altering the exception mask, it reports the callback event `copyfmt_event`. If after the copy is complete, `state&exceptions` is nonzero, the function effectively calls clear with the argument rdstate. It returns **`*this`**.

Example

```
C++
```

```
// basic_ios_copyfmt.cpp
// compile with: /EHsc
#include <iostream>
```

```
#include <fstream>

int main( )
{
    using namespace std;
    ofstream x( "test.txt" );
    int i = 10;

    x << showpos;
    cout << i << endl;
    cout.copyfmt( x );
    cout << i << endl;
}
```

`basic_ios::eof`

Indicates if the end of a stream has been reached.

C++

```
bool eof() const;
```

Return Value

`true` if the end of the stream has been reached, `false` otherwise.

Remarks

The member function returns `true` if rdstate `& eofbit` is nonzero. For more information on `eofbit`, see ios_base::iostate.

Example

C++

```
// basic_ios_eof.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char* argv[] )
{
    fstream   fs;
```

```
    int n = 1;
    fs.open( "basic_ios_eof.txt" );   // an empty file
    cout << boolalpha
    cout << fs.eof() << endl;
    fs >> n;   // Read the char in the file
    cout << fs.eof() << endl;
}
```

`basic_ios::exceptions`

Indicates which exceptions will be thrown by the stream.

C++

```
iostate exceptions() const;
void exceptions(iostate Newexcept);
void exceptions(io_state Newexcept);
```

Parameters

`Newexcept`

The flags that you want to throw an exception.

Return Value

The flags that are currently specified to throw an exception for the stream.

Remarks

The first member function returns the stored exception mask. The second member function stores *`_Except`* in the exception mask and returns its previous stored value. Storing a new exception mask can throw an exception just like the call clear(rdstate).

Example

C++

```
// basic_ios_exceptions.cpp
// compile with: /EHsc /GR
#include <iostream>

int main( )
{
```

```
    using namespace std;

    cout << cout.exceptions( ) << endl;
    cout.exceptions( ios::eofbit );
    cout << cout.exceptions( ) << endl;
    try
    {
        cout.clear( ios::eofbit );   // Force eofbit on
    }
    catch ( exception &e )
    {
        cout.clear( );
        cout << "Caught the exception." << endl;
        cout << "Exception class: " << typeid(e).name()  << endl;
        cout << "Exception description: " << e.what() << endl;
    }
}
```

Output

```
0
1
Caught the exception.
Exception class: class std::ios_base::failure
Exception description: ios_base::eofbit set
```

`basic_ios::fail`

Indicates failure to extract a valid field from a stream.

C++

```
bool fail() const;
```

Return Value

`true` if `rdstate & (badbit|failbit)` is nonzero, otherwise **`false`**.

For more information on `failbit`, see ios_base::iostate.

Example

C++

```
// basic_ios_fail.cpp
// compile with: /EHsc
```

```
#include <iostream>

int main( void )
{
    using namespace std;
    bool b = cout.fail( );
    cout << boolalpha;
    cout << b << endl;
}
```

`basic_ios::fill`

Specifies or returns the character that will be used when the text isn't as wide as the stream.

C++

```
char_type fill() const;
char_type fill(char_type Char);
```

Parameters

`Char`

The character you want as the fill character.

Return Value

The current fill character.

Remarks

The first member function returns the stored fill character. The second member function stores *`Char`* in the fill character and returns its previous stored value.

Example

C++

```
// basic_ios_fill.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>
```

```
int main( )
{
    using namespace std;

    cout << setw( 5 ) << 'a' << endl;
    cout.fill( 'x' );
    cout << setw( 5 ) << 'a' << endl;
    cout << cout.fill( ) << endl;
}
```

Output

```
a
xxxxa
x
```

`basic_ios::good`

Indicates the stream is in good condition.

C++

```
bool good() const;
```

Return Value

`true` if `rdstate == goodbit` (no state flags are set), otherwise, `false`.

For more information on `goodbit`, see ios_base::iostate.

Example

See basic_ios::bad for an example of using `good`.

`basic_ios::imbue`

Changes the locale.

C++

```
locale imbue(const locale& Loc);
```

Parameters

Loc

A locale string.

Return Value

The previous locale.

Remarks

If rdbuf isn't a `NULL` pointer, the member function calls

`rdbuf-> pubimbue(_ Loc)`

In any case, it returns `ios_base::imbue(_ Loc)`.

For more information, see pubimbue and ios_base::imbue.

Example

C++

```
// basic_ios_imbue.cpp
// compile with: /EHsc
#include <iostream>
#include <locale>

int main( )
{
    using namespace std;

    cout.imbue( locale( "french_france" ) );
    double x = 1234567.123456;
    cout << x << endl;
}
```

`basic_ios::init`

Called by `basic_ios` constructors.

C++

```
void init(basic_streambuf<Elem,Traits>* _Sb, bool _Isstd = false);
```

Parameters

`_Sb`
Standard buffer to store input or output elements.

`_Isstd`
Specifies whether this is a standard stream.

Remarks

The member function stores values in all member objects, so that:

- rdbuf returns `_Sb`.
- tie returns a `NULL` pointer.
- rdstate returns goodbit if `_Sb` is nonzero; otherwise, it returns badbit.
- exceptions returns `goodbit`.
- flags returns `skipws | dec`. For more information, see skipws and dec.
- width returns 0.
- precision returns 6.
- fill returns the space character.
- getloc returns `locale::classic`.
- iword returns zero, and pword returns a `NULL` pointer for all argument values.

`basic_ios::int_type`

A synonym for `traits_type::int_type`.

```
C++

typedef typename traits_type::int_type int_type;
```

`basic_ios::move`

Moves all values, except the pointer to the stream buffer, from the parameter to the current object.

```
C++
```

```
void move(basic_ios&& right);
```

Parameters

right

The ios_base object to move values from.

Remarks

The protected member function moves all the values stored in *right* to ***this** except the stored stream buffer pointer, which is unchanged in *right* and set to a NULL pointer in ***this**. The stored tie pointer is set to a NULL pointer in *right*.

basic_ios::narrow

Finds the equivalent char to a given char_type.

```
C++
```

```
char narrow(char_type Char, char Default = '\0') const;
```

Parameters

Char

The **char** to convert.

Default

The **char** that you want returned if no equivalent is found.

Return Value

The equivalent **char** to a given char_type.

Remarks

The member function returns `use_facet<ctype<E>>(getloc()).narrow(Char, Default)`.

For more information, see use_facet and getloc.

Example

```
C++
```

```
// basic_ios_narrow.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
    using namespace std;
    wchar_t *x = L"test";
    char y[10];
    cout << x[0] << endl;
    wcout << x << endl;
    y[0] = wcout.narrow( x[0] );
    cout << y[0] << endl;
}
```

`basic_ios::off_type`

A synonym for `traits_type::off_type`.

```
C++
```

```
typedef typename traits_type::off_type off_type;
```

`basic_ios::operator void *`

Indicates if the stream is still good.

```
C++
```

```
operator void *() const;
```

Return Value

The operator returns a `NULL` pointer only if fail.

Example

C++

```
// basic_ios_opgood.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << (bool)(&cout != 0) << endl;   // Stream is still good
}
```

Output

```
1
```

`basic_ios::operator!`

Indicates if the stream isn't bad.

C++

```
bool operator!() const;
```

Return Value

Returns the same as calling basic_ios::fail

Example

C++

```
// basic_ios_opbad.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
```

```
    cout << !cout << endl;   // Stream is not bad
}
```

Output

```
0
```

`basic_ios::operator bool`

Allows use of a `basic_ios` object as a **`bool`**. Automatic type conversion is disabled to prevent common, unintended side effects.

C++

```
explicit operator bool() const;
```

Remarks

Returns `true` if the stream has no errors; otherwise `false`.

`basic_ios::pos_type`

A synonym for `traits_type::pos_type`.

C++

```
typedef typename traits_type::pos_type pos_type;
```

`basic_ios::rdbuf`

Routes stream to specified buffer.

C++

```
basic_streambuf<Elem, Traits> *rdbuf() const;
basic_streambuf<Elem, Traits> *rdbuf(
basic_streambuf<Elem, Traits>* _Sb);
```

Parameters

_Sb

A stream.

Remarks

The first member function returns the stored stream buffer pointer.

The second member function stores *_Sb* in the stored stream buffer pointer and returns the previously stored value.

Example

C++

```
// basic_ios_rdbuf.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;
    ofstream file( "rdbuf.txt" );
    streambuf *x = cout.rdbuf( file.rdbuf( ) );
    cout << "test" << endl;   // Goes to file
    cout.rdbuf(x);
    cout << "test2" << endl;
}
```

Output

```
test2
```

basic_ios::rdstate

Reads the stream error state.

C++

```
iostate rdstate() const;
```

Return Value

The stored stream state information.

Example

C++

```
// basic_ios_rdstate.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>
using namespace std;

void TestFlags( ios& x )
{
    cout << ( x.rdstate( ) & ios::badbit ) << endl;
    cout << ( x.rdstate( ) & ios::failbit ) << endl;
    cout << ( x.rdstate( ) & ios::eofbit ) << endl;
    cout << endl;
}

int main( )
{
    fstream x( "c:\test.txt", ios::out );
    x.clear( );
    TestFlags( x );
    x.clear( ios::badbit | ios::failbit | ios::eofbit );
    TestFlags( x );
}
```

Output

```
0
0
0

4
2
1
```

`basic_ios::setstate`

Sets the specified stream error flags (currently set stream error state flags remain unchanged):

flag	Description
goodbit	No error

flag	Description
badbit	Unrecoverable error
failbit	I/O operation failed, such as a formatting or extraction error
eofbit	Stream has reached the end of the file

```
C++
```

```
void setstate(iostate _State);
```

Parameters

`_State`

Additional flags to set.

Remarks

The member function essentially calls `clear(_ state | rdstate)`.

For more information, see clear and rdstate.

Example

```
C++
```

```
// basic_ios_setstate.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
using namespace std;

int main( )
{
    bool b = cout.bad( );
    cout << b << endl;   // Good
    cout.clear( ios::badbit );
    b = cout.bad( );
    // cout.clear( );
    cout << b << endl;   // Is bad, good
    b = cout.fail( );
    cout << b << endl;   // Not failed
    cout.setstate( ios::failbit );
    b = cout.fail( );
    cout.clear( );
    cout << b << endl;   // Is failed, good
```

```
    return 0;
}
```

Output

```
0
1
```

`basic_ios::set_rdbuf`

Assigns a stream buffer to be the read buffer for this stream object.

C++

```
void set_rdbuf(
basic_streambuf<Elem, Tr>* strbuf)
```

Parameters

`strbuf`
The stream buffer to become the read buffer.

Remarks

The protected member function stores *`strbuf`* in the stream buffer pointer. It doesn't call `clear`.

`basic_ios::tie`

Ensures that one stream is processed before another stream.

C++

```
basic_ostream<Elem, Traits> *tie() const;
basic_ostream<Elem, Traits> *tie(
basic_ostream<Elem, Traits>* str);
```

Parameters

str

A stream.

Return Value

The first member function returns the stored tie pointer. The second member function stores *str* in the tie pointer and returns its previous stored value.

Remarks

`tie` causes two streams to be synchronized, such that, operations on one stream occur after operations on the other stream are complete.

Example

In the following example, by tying `cin` to `cout`, it's guaranteed that the `Enter a number:` string will go to the console before the number itself is extracted from `cin`. This eliminates the possibility that the "Enter a number:" string is still sitting in the buffer when the number is read, so that we're certain that the user actually has some prompt to respond to. By default, `cin` and `cout` are tied.

C++

```
#include <ios>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    cin.tie( &cout );
    cout << "Enter a number:";
    cin >> i;
}
```

`basic_ios::traits_type`

A synonym for the template parameter `Traits`.

C++

```
typedef Traits traits_type;
```

`basic_ios::widen`

Finds the equivalent `char_type` to a given `char`.

```
C++
```

```
char_type widen(char Char) const;
```

Parameters

`Char`

The character to convert.

Return Value

Finds the equivalent `char_type` to a given `char`.

Remarks

The member function returns `use_facet<ctype<E>>(getloc).widen(Char)`.

For more information, see use_facet and getloc.

Example

```
C++
```

```
// basic_ios_widen.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <wchar.h>

int main( )
{
    using namespace std;
    char *z = "Hello";
    wchar_t y[2] = {0,0};
    cout << z[0] << endl;
    y[0] = wcout.widen( z[0] );
    wcout << &y[0] << endl;
}
```

`basic_ios::swap`

Exchanges the values in this `basic_ios` object for those of another `basic_ios` object. However, the pointers to the stream buffers aren't swapped.

```
C++
```

```
void swap(basic_ios&& right);
```

Parameters

`right`

The `basic_ios` object that is used to exchange values.

Remarks

The protected member function exchanges all the values stored in *`right`* with **`*this`** except the stored `stream buffer pointer`.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

fpos Class

Article • 06/17/2022

The class template describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream. An object of class fpos< **St**> effectively stores at least two member objects:

- A byte offset, of type streamoff.
- A conversion state, for use by an object of class basic_filebuf, of type `St`, typically `mbstate_t`.

It can also store an arbitrary file position, for use by an object of class basic_filebuf, of type `fpos_t`. For an environment with limited file size, however, `streamoff` and `fpos_t` may sometimes be used interchangeably. For an environment with no streams that have a state-dependent encoding, `mbstate_t` may actually be unused. Therefore, the number of member objects stored may vary.

Syntax

C++

```
template <class Statetype>
class fpos
```

Parameters

Statetype
State information.

Constructors

Constructor	Description
fpos	Create an object that contains information about a position (offset) in a stream.

Member functions

Member function	Description
seekpos	Used internally by the C++ Standard Library only. Don't call this method from your code.
state	Sets or returns the conversion state.

Operators

Operator	Description
operator!=	Tests file-position indicators for inequality.
operator+	Increments a file-position indicator.
operator+=	Increments a file-position indicator.
operator-	Decrements a file-position indicator.
operator-=	Decrements a file-position indicator.
operator==	Tests file-position indicators for equality.
operator streamoff	Casts object of type `fpos` to object of type `streamoff`.

Requirements

Header: <ios>

Namespace: std

fpos::fpos

Create an object that contains information about a position (offset) in a stream.

```
C++
```

```
fpos(streamoff _Off = 0);

fpos(Statetype _State, fpos_t _Filepos);
```

Parameters

_Off
The offset into the stream.

_State
The starting state of the `fpos` object.

_Filepos
The offset into the stream.

Remarks

The first constructor stores the offset *_Off*, relative to the beginning of file and in the initial conversion state. If *_Off* is -1, the resulting object represents an invalid stream position.

The second constructor stores a zero offset and the object *_State*.

fpos::operator!=

Tests file-position indicators for inequality.

```
C++
```

```
bool operator!=(const fpos<Statetype>& right) const;
```

Parameters

right
The file-position indicator against which to compare.

Return Value

`true` if the file-position indicators aren't equal, otherwise **`false`**.

Remarks

The member function returns `!(*this == right)`.

Example

```
C++
```

```cpp
// fpos_op_neq.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main( )
{
   using namespace std;

   fpos<int> pos1, pos2;
   ifstream file;
   char c;

   // Compare two fpos object
   if ( pos1 != pos2 )
      cout << "File position pos1 and pos2 are not equal" << endl;
   else
      cout << "File position pos1 and pos2 are equal" << endl;

   file.open( "fpos_op_neq.txt" );
   file.seekg( 0 );   // Goes to a zero-based position in the file
   pos1 = file.tellg( );
   file.get( c);
   cout << c << endl;

   // Increment pos1
   pos1 += 1;
   file.get( c );
   cout << c << endl;

   pos1 = file.tellg( ) - fpos<int>( 2);
   file.seekg( pos1 );
   file.get( c );
   cout << c << endl;

   // Increment pos1
   pos1 = pos1 + fpos<int>( 1 );
   file.get(c);
   cout << c << endl;

   pos1 -= fpos<int>( 2 );
   file.seekg( pos1 );
   file.get( c );
   cout << c << endl;

   file.close( );
}
```

fpos::operator+

Increments a file-position indicator.

```
C++
```

```
fpos<Statetype> operator+(streamoff _Off) const;
```

Parameters

_Off
The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function returns `fpos(*this) +=_Off`.

Example

See operator!= for a sample of using `operator+`.

fpos::operator+=

Increments a file-position indicator.

```
C++
```

```
fpos<Statetype>& operator+=(streamoff _Off);
```

Parameters

_Off
The offset by which you want to increment the file-position indicator.

Return Value

The position in the file.

Remarks

The member function adds *_Off* to the stored offset member object and then returns `*this`. When working with files, the result is valid only for binary streams that don't have a state-dependent encoding.

Example

See operator!= for a sample of using `operator+=`.

fpos::operator-

Decrements a file-position indicator.

```
C++
```

```
streamoff operator-(const fpos<Statetype>& right) const;

fpos<Statetype> operator-(streamoff _Off) const;
```

Parameters

right
File position.

_Off
Stream offset.

Return Value

The first member function returns `(streamoff)*this - (streamoff) right`. The second member function returns `fpos(*this) -= _Off`.

Example

See operator!= for a sample of using `operator-`.

fpos::operator-=

Decrements a file-position indicator.

```
C++
```

```
fpos<Statetype>& operator-=(streamoff _Off);
```

Parameters

_Off
Stream offset.

Return Value

The member function returns `fpos(*this) -= _Off`.

Remarks

When working with files, the result is valid only for binary streams that don't have a state-dependent encoding.

Example

See operator!= for a sample of using `operator-=`.

fpos::operator==

Tests file-position indicators for equality.

```
C++
```

```
bool operator==(const fpos<Statetype>& right) const;
```

Parameters

right
The file-position indicator against which to compare.

Return Value

`true` if the file-position indicators are equal; otherwise **`false`**.

Remarks

The member function returns `(streamoff)*this == (streamoff)right`.

Example

See operator!= for a sample of using `operator+=`.

fpos::operator streamoff

Cast object of type `fpos` to object of type `streamoff`.

C++

```
operator streamoff() const;
```

Remarks

The member function returns the stored offset member object and any additional offset stored as part of the `fpos_t` member object.

Example

C++

```
// fpos_op_streampos.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   streamoff s;
   ofstream file( "rdbuf.txt");

   fpos<mbstate_t> f = file.tellp( );
   // Is equivalent to ..
   // streampos f = file.tellp( );
   s = f;
   cout << s << endl;
}
```

Output

```
0
```

fpos::seekpos

This method is used internally by the C++ Standard Library only. Don't call this method from your code.

```
C++
```

```
fpos_t seekpos() const;
```

fpos::state

Sets or returns the conversion state.

```
C++
```

```
Statetype state() const;

void state(Statetype _State);
```

Parameters

_State
The new conversion state.

Return Value

The conversion state.

Remarks

The first member function returns the value stored in the `St` member object. The second member function stores *_State* in the `St` member object.

Example

```
C++
```

```
// fpos_state.cpp
// compile with: /EHsc
#include <ios>
#include <iostream>
#include <fstream>

int main() {
   using namespace std;
   streamoff s;
   ifstream file( "fpos_state.txt" );

   fpos<mbstate_t> f = file.tellg( );
   char ch;
   while ( !file.eof( ) )
      file.get( ch );
   s = f;
   cout << f.state( ) << endl;
   f.state( 9 );
   cout << f.state( ) << endl;
}
```

See also

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

`ios_base` Class

Article • 06/17/2022

The class describes the storage and member functions common to both input and output streams that don't depend on the template parameters. (The class template basic_ios describes what is common and is dependent on template parameters.)

An object of class `ios_base` stores formatting information, which consists of:

- Format flags in an object of type fmtflags.
- An exception mask in an object of type iostate.
- A field width in an object of type `int`.
- A display precision in an object of type `int`.
- A `locale` object in an object of type `locale`.
- Two extensible arrays, with elements of type `long` and `void` pointer.

An object of class `ios_base` also stores stream state information, in an object of type iostate, and a callback stack.

Members

Constructors

Name	Description
ios_base	Constructs `ios_base` objects.

Typedefs

Name	Description
event_callback	Describes a function passed to register_call.
fmtflags	Constants to specify the appearance of output.
iostate	Defines constants describing the state of a stream.
openmode	Describes how to interact with a stream.

Name	Description
seekdir	Specifies starting point for offset operations.

Enums

Name	Description
event	Specifies event types.

Constants

Name	Description
adjustfield	A bitmask defined as `internal` | `left` | `right`.
app	Specifies seeking to the end of a stream before each insertion.
ate	Specifies seeking to the end of a stream when its controlling object is first created.
badbit	Records a loss of integrity of the stream buffer.
basefield	A bitmask defined as `dec` | `hex` | `oct`.
beg	Specifies seeking relative to the beginning of a sequence.
binary	Specifies that a file should be read as a binary stream, rather than as a text stream.
boolalpha	Specifies insertion or extraction of objects of type `bool` as names (such as `true` and `false`) rather than as numeric values.
cur	Specifies seeking relative to the current position within a sequence.
dec	Specifies insertion or extraction of integer values in decimal format.
end	Specifies seeking relative to the end of a sequence.
eofbit	Records end-of-file while extracting from a stream.
failbit	Records a failure to extract a valid field from a stream.
fixed	Specifies insertion of floating-point values in fixed-point format (with no exponent field).
floatfield	A bitmask defined as `fixed` | `scientific`
goodbit	All state bits clear.
hex	Specifies insertion or extraction of integer values in hexadecimal format.

Name	Description
in	Specifies extraction from a stream.
internal	Pads to a field width by inserting fill characters at a point internal to a generated numeric field.
left	Specifies left justification.
oct	Specifies insertion or extraction of integer values in octal format.
out	Specifies insertion to a stream.
right	Specifies right justification.
scientific	Specifies insertion of floating-point values in scientific format (with an exponent field).
showbase	Specifies insertion of a prefix that reveals the base of a generated integer field.
showpoint	Specifies unconditional insertion of a decimal point in a generated floating-point field.
showpos	Specifies insertion of a plus sign in a nonnegative generated numeric field.
skipws	Specifies skipping leading white space before certain extractions.
trunc	Specifies deleting contents of an existing file when its controlling object is created.
unitbuf	Causes output to be flushed after each insertion.
uppercase	Specifies insertion of uppercase equivalents of lowercase letters in certain insertions.

Functions

Name	Description
failure	The member class serves as the base class for all exceptions thrown by the member function clear in class template basic_ios.
flags	Sets or returns the current flag settings.
getloc	Returns the stored `locale` object.
imbue	Changes the locale.
Init	Creates the standard `iostream` objects when constructed.
iword	Assigns a value to be stored as an `iword`.
precision	Specifies the number of digits to display in a floating-point number.

Name	Description
pword	Assigns a value to be stored as a `pword`.
register_callback	Specifies a callback function.
setf	Sets the specified flags.
sync_with_stdio	Ensures that `iostream` and C run-time library operations occur in the order that they appear in source code.
unsetf	Causes the specified flags to be off.
width	Sets the length of the output stream.
xalloc	Specifies that a variable shall be part of the stream.

Operators

Name	Description
operator=	The assignment operator for `ios_base` objects.

Requirements

Header: `<ios>`

Namespace: `std`

`event`

Specifies event types.

```
C++
```

```
enum event {
    erase_event,
    imbue_event,
    copyfmt_event};
```

Remarks

The type is an enumerated type that describes an object that can store the callback event used as an argument to a function registered with register_callback. The distinct

event values are:

- `copyfmt_event`, to identify a callback that occurs near the end of a call to copyfmt, just before the exception mask is copied.
- `erase_event`, to identify a callback that occurs at the beginning of a call to copyfmt, or at the beginning of a call to the destructor for `*this`.
- `imbue_event`, to identify a callback that occurs at the end of a call to imbue, just before the function returns.

Example

See register_callback for an example.

`event_callback`

Describes a function passed to register_call.

C++

```
typedef void (__cdecl *event_callback)(
    event _E,
    ios_base& _Base,
    int _I);
```

Parameters

`_E`
The event.

`_Base`
The stream in which the event was called.

`_I`
A user-defined number.

Remarks

The type describes a pointer to a function that can be registered with register_callback. This type of function must not throw an exception.

Example

See register_call for an example that uses `event_callback`.

`failure`

The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the `iostreams` library, to report errors detected during stream buffer operations.

C++

```
namespace std {
    class failure : public system_error {
    public:
        explicit failure(
            const string& _Message,
            const error_code& _Code = io_errc::stream);

        explicit failure(
            const char* str,
            const error_code& _Code = io_errc::stream);
    };
}
```

Remarks

The value returned by `what()` is a copy of `_Message`, possibly augmented with a test based on `_Code`. If `_Code` isn't specified, the default value is `make_error_code(io_errc::stream)`.

Example

C++

```
// ios_base_failure.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.exceptions(ios::failbit);
```

```
    try
    {
        file.open( "rm.txt", ios_base::in );
        // Opens nonexistent file for reading
    }
    catch( ios_base::failure f )
    {
        cout << "Caught an exception: " << f.what() << endl;
    }
}
```

Output

```
Caught an exception: ios_base::failbit set
```

`flags`

Sets or returns the current flag settings.

C++

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

Parameters

`fmtfl`

The new `fmtflags` setting.

Return Value

The previous or current `fmtflags` setting.

Remarks

See ios_base::fmtflags for a list of the flags.

The first member function returns the stored format flags. The second member function stores *`fmtfl`* in the format flags and returns its previous stored value.

Example

C++

```
// ios_base_flags.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    cout << cout.flags( ) << endl;
    cout.flags( ios::dec | ios::boolalpha );
    cout << cout.flags( );
}
```

Output

```
513
16896
```

fmtflags

Constants to specify the appearance of output.

C++

```
class ios_base {
public:
   typedef implementation-defined-bitmask-type fmtflags;
   static const fmtflags boolalpha;
   static const fmtflags dec;
   static const fmtflags fixed;
   static const fmtflags hex;
   static const fmtflags internal;
   static const fmtflags left;
   static const fmtflags oct;
   static const fmtflags right;
   static const fmtflags scientific;
   static const fmtflags showbase;
   static const fmtflags showpoint;
   static const fmtflags showpos;
   static const fmtflags skipws;
   static const fmtflags unitbuf;
   static const fmtflags uppercase;
   static const fmtflags adjustfield;
   static const fmtflags basefield;
   static const fmtflags floatfield;
   // ...
};
```

Remarks

Supports the manipulators in ios.

The type is a bitmask type that describes an object that can store format flags. The distinct flag values (elements) are:

- `dec`, to insert or extract integer values in decimal format.
- `hex`, to insert or extract integer values in hexadecimal format.
- `oct`, to insert or extract integer values in octal format.
- `showbase`, to insert a prefix that reveals the base of a generated integer field.
- `internal`, to pad to a field width as needed by inserting fill characters at a point internal to a generated numeric field. (For information on setting the field width, see setw).
- `left`, to pad to a field width as needed by inserting fill characters at the end of a generated field (left justification).
- `right`, to pad to a field width as needed by inserting fill characters at the beginning of a generated field (right justification).
- `boolalpha`, to insert or extract objects of type `bool` as names (such as `true` and `false`) rather than as numeric values.
- `fixed`, to insert floating-point values in fixed-point format (with no exponent field).
- `scientific`, to insert floating-point values in scientific format (with an exponent field).
- `showpoint`, to insert a decimal point unconditionally in a generated floating-point field.
- `showpos`, to insert a plus sign in a nonnegative generated numeric field.
- `skipws`, to skip leading white space before certain extractions.
- `unitbuf`, to flush output after each insertion.
- `uppercase`, to insert uppercase equivalents of lowercase letters in certain insertions.

In addition, several useful values are:

- `adjustfield`, a bitmask defined as `internal` | `left` | `right`
- `basefield`, defined as `dec` | `hex` | `oct`
- `floatfield`, defined as `fixed` | `scientific`

For examples of functions that modify these format flags, see <iomanip>.

`getloc`

Returns the stored `locale` object.

C++

```
locale getloc() const;
```

Return Value

The stored `locale` object.

Example

C++

```
// ios_base_getlock.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    cout << cout.getloc( ).name( ).c_str( ) << endl;
}
```

Output

```
C
```

`imbue`

Changes the locale.

```
C++
```

```
locale imbue(const locale& _Loc);
```

Parameters

_Loc
The new locale setting.

Return Value

The previous locale.

Remarks

The member function stores *_Loc* in the `locale` object and then reports the callback event and `imbue_event`. It returns the previous stored value.

Example

See basic_ios::imbue for a sample.

`Init`

Creates the standard `iostream` objects when constructed.

```
C++
```

```
class Init { };
```

Remarks

The nested class describes an object whose construction ensures that the standard `iostream` objects are properly constructed, even before the execution of a constructor for an arbitrary static object.

`ios_base`

Constructs `ios_base` objects.

```
C++

ios_base();
```

Remarks

The (protected) constructor does nothing. A later call to `basic_ios::`init must initialize the object before it can be safely destroyed. Thus, the only safe use for class `ios_base` is as a base class for class template basic_ios.

`iostate`

The type of constants that describe the state of a stream.

```
C++

class ios_base {
public:
   typedef implementation-defined-bitmask-type iostate;
   static const iostate badbit;
   static const iostate eofbit;
   static const iostate failbit;
   static const iostate goodbit;
   // ...
};
```

Remarks

The type is a bitmask type that describes an object that can store stream state information. The distinct flag values (elements) are:

- `badbit`, to record a loss of integrity of the stream buffer.
- `eofbit`, to record end-of-file while extracting from a stream.
- `failbit`, to record a failure to extract a valid field from a stream.

In addition, a useful value is `goodbit`, where none of the previously mentioned bits are set (`goodbit` is guaranteed to be zero).

`iword`

Assigns a value to be stored as an `iword`.

```
C++
```

```
long& iword(int idx);
```

Parameters

idx
The index of the value to store as an `iword`.

Remarks

The member function returns a reference to element *idx* of the extensible array with elements of type `long`. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to `iword` for the object, after the object is altered by a call to `basic_ios::`copyfmt, or after the object is destroyed.

If *idx* is negative or if unique storage is unavailable for the element, the function calls setstate`(badbit)` and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call xalloc.

Example

See xalloc for a sample of how to use `iword`.

openmode

Describes how to interact with a stream.

```
C++
```

```
class ios_base {
public:
   typedef implementation-defined-bitmask-type openmode;
   static const openmode  in;
   static const openmode  out;
   static const openmode  ate;
   static const openmode  app;
   static const openmode  trunc;
   static const openmode  binary;
```

```
    // ...
};
```

Remarks

The opening mode for several `iostream` objects. The flag values are:

Constant	Effect
`app`	Seek to the end of the stream before each write
`ate`	Seek to the end of the stream immediately after open
`binary`	Open in binary mode. (See fopen for a description of binary mode.)
`in`	Open for reading
`out`	Open for writing
`trunc`	Delete the contents of the file after open

Example

C++

```
// ios_base_openmode.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.open( "rm.txt", ios_base::out | ios_base::trunc );

    file << "testing";
}
```

`operator=`

The assignment operator for `ios_base` objects.

C++

```
ios_base& operator=(const ios_base& right);
```

Parameters

right
An object of type `ios_base`.

Return Value

The object being assigned to.

Remarks

The operator copies the stored formatting information, making a new copy of any extensible arrays. It then returns `*this`. The callback stack isn't copied.

This operator is only used by classes derived from `ios_base`.

`precision`

Specifies the number of digits to display in a floating-point number.

C++

```
streamsize precision() const;
streamsize precision(streamsize _Prec);
```

Parameters

_Prec
The number of significant digits to display, or the number of digits after the decimal point in fixed notation.

Return Value

The first member function returns the stored display precision. The second member function stores *_Prec* in the display precision and returns its previous stored value.

Remarks

Floating-point numbers are displayed in fixed notation with fixed.

Example

C++

```
// ios_base_precision.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    float i = 31.31234F;

    cout.precision( 3 );
    cout << i << endl;          // display three significant digits
    cout << fixed << i << endl; // display three digits after decimal
                                // point
}
```

Output

```
31.3
31.312
```

pword

Assigns a value to be stored as a `pword`.

C++

```
void *& pword(int index);
```

Parameters

index

The index of the value to store as a `pword`.

Remarks

The member function returns a reference to element *index* of the extensible array with elements of type `void` pointer. All elements are effectively present and initially store the

null pointer. The returned reference is invalid after the next call to `pword` for the object, after the object is altered by a call to `basic_ios::`copyfmt, or after the object is destroyed.

If *index* is negative, or if unique storage is unavailable for the element, the function calls setstate`(badbit)` and returns a reference that might not be unique.

To obtain a unique index, for use across all objects of type `ios_base`, call xalloc.

Example

See xalloc for an example of using `pword`.

`register_callback`

Specifies a callback function.

C++

```
void register_callback(
    event_callback pfn, int idx);
```

Parameters

`pfn`
Pointer to the callback function.

`idx`
A user-defined number.

Remarks

The member function pushes the pair `{pfn, idx}` onto the stored callback stack callback stack. When a callback event **ev** is reported, the functions are called, in reverse order of registry, by the expression `(*pfn)(ev, *this, idx)`.

Example

C++

```cpp
// ios_base_register_callback.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

void callback1( ios_base::event e, ios_base& stream, int arg )
{
    cout << "in callback1" << endl;
    switch ( e )
    {
    case ios_base::erase_event:
        cout << "an erase event" << endl;
        break;
    case ios_base::imbue_event:
        cout << "an imbue event" << endl;
        break;
    case ios_base::copyfmt_event:
        cout << "an copyfmt event" << endl;
        break;
    };
}

void callback2( ios_base::event e, ios_base& stream, int arg )
{
    cout << "in callback2" << endl;
    switch ( e )
    {
    case ios_base::erase_event:
        cout << "an erase event" << endl;
        break;
    case ios_base::imbue_event:
        cout << "an imbue event" << endl;
        break;
    case ios_base::copyfmt_event:
        cout << "an copyfmt event" << endl;
        break;
    };
}

int main( )
{
    // Make sure the imbue will not throw an exception
    // assert( setlocale( LC_ALL, "german" )!=NULL );

    cout.register_callback( callback1, 0 );
    cin.register_callback( callback2, 0 );

    try
    {
        // If no exception because the locale's not found,
        // generate an imbue_event on callback1
        cout.imbue(locale("german"));
```

```
    }
    catch(...)
    {
        cout << "exception" << endl;
    }

    // This will
    // (1) erase_event on callback1
    // (2) copyfmt_event on callback2
    cout.copyfmt(cin);

    // We get two erase events from callback2 at the end because
    // both cin and cout have callback2 registered when cin and cout
    // are destroyed at the end of program.
}
```

Output

```
in callback1
an imbue event
in callback1
an erase event
in callback2
an copyfmt event
in callback2
an erase event
in callback2
an erase event
```

`seekdir`

Specifies starting point for offset operations.

C++

```
namespace std {
    class ios_base {
    public:
        typedef implementation-defined-enumerated-type seekdir;
        static const seekdir beg;
        static const seekdir cur;
        static const seekdir end;
        // ...
    };
}
```

Remarks

The type is an enumerated type that describes an object that can store the seek mode used as an argument to the member functions of several `iostream` classes. The distinct flag values are:

- `beg`, to seek (alter the current read or write position) relative to the beginning of a sequence (array, stream, or file).
- `cur`, to seek relative to the current position within a sequence.
- `end`, to seek relative to the end of a sequence.

Example

C++

```
// ios_base_seekdir.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
    using namespace std;
    fstream file;
    file.open( "rm.txt", ios_base::out | ios_base::trunc );

    file << "testing";
    file.seekp( 0, ios_base::beg );
    file << "a";
    file.seekp( 0, ios_base::end );
    file << "a";
}
```

`setf`

Sets the specified flags.

C++

```
fmtflags setf(
    fmtflags _Mask
);
fmtflags setf(
    fmtflags _Mask,
    fmtflags _Unset
);
```

Parameters

_Mask

The flags to turn on.

_Unset

The flags to turn off.

Return Value

The previous format flags

Remarks

The first member function effectively calls flags(_Mask | _Flags) (set selected bits) and then returns the previous format flags. The second member function effectively calls `flags(_Mask & fmtfl, flags & ~_Mask)` (replace selected bits under a mask) and then returns the previous format flags.

Example

C++

```
// ios_base_setf.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
    using namespace std;
    int i = 10;
    cout << i << endl;

    cout.unsetf( ios_base::dec );
    cout.setf( ios_base::hex );
    cout << i << endl;

    cout.setf( ios_base::dec );
    cout << i << endl;
    cout.setf( ios_base::hex, ios_base::dec );
    cout << i << endl;
}
```

sync_with_stdio

Ensures that `iostream` and C run-time library operations occur in the order that they appear in source code.

```
C++
```

```
static bool sync_with_stdio(
   bool _Sync = true
);
```

Parameters

`_Sync`
Whether all streams are in sync with `stdio`.

Return Value

Previous setting for this function.

Remarks

The static member function stores a `stdio` sync flag, which is initially `true`. When `true`, this flag ensures that operations on the same file are properly synchronized between the iostreams functions and functions defined in the C++ Standard Library. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores *`_Sync`* in the `stdio` sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

`unsetf`

Turns off the specified flags.

```
C++
```

```
void unsetf(
   fmtflags _Mask
);
```

Parameters

_Mask

The flags that you want to turn off.

Remarks

The member function effectively calls flags(~_Mask & flags) (clear selected bits).

Example

See ios_base::setf for a sample of using `unsetf`.

`width`

Sets the length of the output stream.

```
C++
```

```
streamsize width( ) const;
streamsize width(
   streamsize _Wide
);
```

Parameters

_Wide

The desired size of the output stream.

Return Value

The current width setting.

Remarks

The first member function returns the stored field width. The second member function stores *_Wide* in the field width and returns its previous stored value.

Example

```
C++
```

```
// ios_base_width.cpp
// compile with: /EHsc
#include <iostream>

int main( ) {
    using namespace std;

    cout.width( 20 );
    cout << cout.width( ) << endl;
    cout << cout.width( ) << endl;
}
```

Output

```
20
0
```

xalloc

Specifies that a variable is part of the stream.

C++

```
static int xalloc( );
```

Return Value

The static member function returns a stored static value, which it increments on each call.

Remarks

You can use the return value as a unique index argument when calling the member functions iword or pword.

Example

C++

```
// ios_base_xalloc.cpp
// compile with: /EHsc
// Lets you store user-defined information.
// iword, jword, xalloc
```

```cpp
#include <iostream>

int main( )
{
    using namespace std;

    static const int i = ios_base::xalloc();
    static const int j = ios_base::xalloc();
    cout.iword( i ) = 11;
    cin.iword( i ) = 13;
    cin.pword( j ) = "testing";
    cout << cout.iword( i ) << endl;
    cout << cin.iword( i ) << endl;
    cout << ( char * )cin.pword( j ) << endl;
}
```

Output

```
11
13
testing
```

See also

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

<iosfwd>

Article • 12/07/2021

Declares forward references to several class templates used throughout iostreams. All such class templates are defined in other standard headers. You include this header explicitly only when you need one of its declarations, but not its definition.

Syntax

C++

```
#include <iosfwd>
```

Typedefs

C++

```
typedef T1 streamoff;
typedef T2 streamsize;
typedef fpos streampos;

// char TYPE DEFINITIONS
typedef basic_ios<char, char_traits<char>> ios;
typedef basic_streambuf<char, char_traits<char>> streambuf;
typedef basic_istream<char, char_traits<char>> istream;
typedef basic_ostream<char, char_traits<char>> ostream;
typedef basic_iostream<char, char_traits<char>> iostream;
typedef basic_stringbuf<char, char_traits<char>> stringbuf;
typedef basic_istringstream<char, char_traits<char>> istringstream;
typedef basic_ostringstream<char, char_traits<char>> ostringstream;
typedef basic_stringstream<char, char_traits<char>> stringstream;
typedef basic_filebuf<char, char_traits<char>> filebuf;
typedef basic_ifstream<char, char_traits<char>> ifstream;
typedef basic_ofstream<char, char_traits<char>> ofstream;
typedef basic_fstream<char, char_traits<char>> fstream;

// wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t>> wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t>> wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t>> wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t>> wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t>> wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t>> wstringbuf;
typedef basic_istringstream<wchar_t, char_traits<wchar_t>> wistringstream;
typedef basic_ostringstream<wchar_t, char_traits<wchar_t>> wostringstream;
typedef basic_stringstream<wchar_t, char_traits<wchar_t>> wstringstream;
```

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t>> wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t>> wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t>> wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t>> wfstream;
```

Forward declarations and class templates

C++

```
template <class _Statetype>
class fpos;

template <class Elem>;
class char_traits;

class char_traits<char>;

class char_traits<wchar_t>;

template <class T>
class allocator;

class ios_base;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ios;

template <class Elem, class Tr = char_traits<Elem>>
class istreambuf_iterator;

template <class Elem, class Tr = char_traits<Elem>>
class ostreambuf_iterator;

template <class Elem, class Tr = char_traits<Elem>>
class basic_streambuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_istream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ostream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_iostream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_stringbuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_istringstream;

template <class Elem, class Tr = char_traits<Elem>>
```

```
class basic_ostringstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_stringstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_filebuf;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ifstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_ofstream;

template <class Elem, class Tr = char_traits<Elem>>
class basic_fstream;
```

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<iostream>`

Article • 12/06/2021

Declares objects that control reading from and writing to the standard streams. This include is often the only header you need to do input and output from a C++ program.

Syntax

```
C++

#include <iostream>
```

> **Note**
>
> The `<iostream>` library uses the `#include <ios>`, `#include <streambuf>`, `#include <istream>`, and `#include <ostream>` statements.

Remarks

The objects fall into two groups:

- cin, cout, cerr, and clog are byte oriented, doing conventional byte-at-a-time transfers.
- wcin, wcout, wcerr, and wclog are wide oriented, translating to and from the wide characters that the program manipulates internally.

Once you do certain operations on a stream, such as the standard input, you can't do operations of a different orientation on the same stream. Therefore, a program can't operate interchangeably on both cin and wcin, for example.

All the objects declared in this header share a peculiar property — you can assume they're constructed before any static objects you define, in a translation unit that includes `<iostream>`. Equally, you can assume that these objects aren't destroyed before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Therefore, you can safely read from or write to the standard streams before program startup and after program termination.

This guarantee isn't universal, however. A static constructor may call a function in another translation unit. The called function can't assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class ios_base::Init.

Global Stream Objects

Name	Description
cerr	Specifies the `cerr` global stream.
cin	Specifies the `cin` global stream.
clog	Specifies the `clog` global stream.
cout	Specifies the `cout` global stream.
wcerr	Specifies the `wcerr` global stream.
wcin	Specifies the `wcin` global stream.
wclog	Specifies the `wclog` global stream.
wcout	Specifies the `wcout` global stream.

`cerr`

The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>`.

```
C++

extern ostream cerr;
```

Return Value

An ostream object.

Remarks

The object controls unbuffered insertions to the standard error output as a byte stream. Once the object is constructed, the expression `cerr.flags & unitbuf` is nonzero, and

`cerr.tie() == &cout`. For more details, see cerr.flags and unitbuf.

Example

C++

```
// iostream_cerr.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

using namespace std;

void TestWide( )
{
   int i = 0;
   wcout << L"Enter a number: ";
   wcin >> i;
   wcerr << L"test for wcerr" << endl;
   wclog << L"test for wclog" << endl;
}

int main( )
{
   int i = 0;
   cout << "Enter a number: ";
   cin >> i;
   cerr << "test for cerr" << endl;
   clog << "test for clog" << endl;
   TestWide( );
}
```

`cin`

Specifies the `cin` global stream.

C++

```
extern istream cin;
```

Return Value

An istream object.

Remarks

The object controls extractions from the standard input as a byte stream. Once the object is constructed, the call cin.tie returns &cout.

Example

In this example, `cin` sets the fail bit on the stream when it comes across non-numeric characters. The program clears the fail bit and strips the invalid character from the stream to continue.

C++

```
// iostream_cin.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
   int x;
   cout << "enter choice:";
   cin >> x;
   while (x < 1 || x > 4)
   {
      cout << "Invalid choice, try again:";
      cin >> x;
      // not a numeric character, probably
      // clear the failure and pull off the non-numeric character
      if (cin.fail())
      {
         cin.clear();
         char c;
         cin >> c;
      }
   }
}
```

Output

```
2
```

`clog`

Specifies the `clog` global stream.

C++

```
extern ostream clog;
```

Return Value

An ostream object.

Remarks

The object controls buffered insertions to the standard error output as a byte stream.

Example

See cerr for an example of using `clog`.

`cout`

Specifies the `cout` global stream.

```
C++

extern ostream cout;
```

Return Value

An ostream object.

Remarks

The object controls insertions to the standard output as a byte stream.

Example

See cerr for an example of using `cout`.

`wcerr`

Specifies the `wcerr` global stream.

```
C++

extern wostream wcerr;
```

Return Value

A wostream object.

Remarks

The object controls unbuffered insertions to the standard error output as a wide stream. Once the object is constructed, the expression `wcerr.flags & unitbuf` is nonzero. For more details, see wcerr.flags and unitbuf.

Example

See cerr for an example of using `wcerr`.

`wcin`

Specifies the `wcin` global stream.

```
C++

extern wistream wcin;
```

Return Value

A wistream object.

Remarks

The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call wcin.tie returns &wcout.

Example

See cerr for an example of using `wcin`.

`wclog`

Specifies the `wclog` global stream.

```
C++
```

```
extern wostream wclog;
```

Return Value

A wostream object.

Remarks

The object controls buffered insertions to the standard error output as a wide stream.

Example

See cerr for an example of using `wclog`.

`wcout`

Specifies the `wcout` global stream.

```
C++
```

```
extern wostream wcout;
```

Return Value

A wostream object.

Remarks

The object controls insertions to the standard output as a wide stream.

Example

See cerr for an example of using `wcout`.

`CString` instances in a `wcout` statement must be cast to `const wchar_t*`, as shown in the following example.

```
C++
```

```
CString cs("meow");

wcout <<(const wchar_t*) cs <<endl;
```

For more information, see Basic CString Operations.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<istream>`

Article • 12/07/2021

Defines the class template basic_istream, which mediates extractions for the iostreams, and the class template basic_iostream, which mediates both insertions and extractions. The header also defines a related manipulator. This header file is typically included for you by another iostreams header; you rarely have to include it directly.

Syntax

```
C++

#include <istream>
```

Typedefs

Type name	Description
iostream	A type `basic_iostream` specialized on `char`.
istream	A type `basic_istream` specialized on `char`.
wiostream	A type `basic_iostream` specialized on **wchar**.
wistream	A type `basic_istream` specialized on **wchar**.

Manipulators

Name	Description
ws	Skips white space in the stream.
swap	Exchanges two stream objects.

Operators

Operator	Description
operator>>	Extracts characters and strings from the stream.

Classes

Class	Description
basic_iostream	A stream class that can do both input and output.
basic_istream	The class template describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type `Elem`, also known as char_type, whose character traits are determined by the class `Tr`, also known as traits_type.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<istream>` functions

Article • 12/07/2021

swap

ws

swap

Exchanges the elements of two stream objects.

C++

```
template <class Elem, class Tr>
void swap(
    basic_istream<Elem, Tr>& left,
    basic_istream<Elem, Tr>& right);

template <class Elem, class Tr>
void swap(
    basic_iostream<Elem, Tr>& left,
    basic_iostream<Elem, Tr>& right);
```

Parameters

left

A stream.

right

A stream.

ws

Skips white space in the stream.

C++

```
template class<Elem, Tr> basic_istream<Elem, Tr>& ws(basic_istream<Elem,
Tr>& _Istr);
```

Parameters

_Istr
A stream.

Return Value

The stream.

Remarks

The manipulator extracts and discards any elements `ch` for which use_facet< **ctype**< **Elem**> >(getloc). **is**(**ctype**< **Elem**>:: **space**, **ch**) is true.

The function calls setstate(**eofbit**) if it encounters end of file while extracting elements. It returns *_Istr*.

Example

See operator>> for an example of using `ws`.

See also

<istream>

`<istream>` operators

Article • 12/07/2021

`operator>>`

Extracts characters and strings from the stream.

C++

```
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr,
    Elem* str);

template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr,
    Elem& Ch);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    signed char* str);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    signed char& Ch);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    unsigned char* str);

template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr,
    unsigned char& Ch);

template <class Elem, class Tr, class Type>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<char, Tr>&& Istr,
    Type& val);
```

Parameters

Ch
A character.

Istr
A stream.

str
A string.

val
A type.

Return Value

The stream

Remarks

The `basic_istream` class also defines several extraction operators. For more information, see basic_istream::operator>>.

The function template:

```
C++
```

```
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr, Elem* str);
```

extracts up to `N - 1` elements and stores them in the array starting at *str*. If `Istr.`width is greater than zero, *N* is `Istr.width`; otherwise, it's the size of the largest array of `Elem` that can be declared. The function always stores the value `Elem()` after any extracted elements it stores. Extraction stops early on end of file, on a character with value `Elem(0)` (which isn't extracted), or on any element (which isn't extracted) that would be discarded by ws. If the function extracts no elements, it calls `Istr.`setstate`(failbit)`. In any case, it calls `Istr.width(0)` and returns *Istr*.

Security Note The null-terminated string being extracted from the input stream must not exceed the size of the destination buffer *str*. For more information, see Avoiding Buffer Overruns.

The function template:

```C++
template <class Elem, class Tr>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<Elem, Tr>& Istr, Elem& Ch);
```

extracts an element, if possible, and stores it in *Ch*. Otherwise, it calls `is.setstate(failbit)`. In any case, it returns *Istr*.

The function template:

```C++
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, signed char* str);
```

returns `Istr >> ( char * ) str`.

The function template:

```C++
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, signed char& Ch);
```

returns `Istr >> ( char& ) Ch`.

The function template:

```C++
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, unsigned char* str);
```

returns `Istr >> ( char * ) str`.

The function template:

```C++
template <class Tr>
basic_istream<char, Tr>& operator>>(
    basic_istream<char, Tr>& Istr, unsigned char& Ch);
```

returns `Istr >> ( char& ) Ch`.

The function template:

```
C++

template <class Elem, class Tr, class Type>
basic_istream<Elem, Tr>& operator>>(
    basic_istream<char, Tr>&& Istr,
    Type& val);
```

returns `Istr >> val` (and converts an rvalue reference to `Istr` to an lvalue in the process).

Example

```
C++

// istream_op_extract.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   ws( cin );
   char c[10];

   cin.width( 9 );
   cin >> c;
   cout << c << endl;
}
```

See also

<istream>

`<istream>` typedefs

Article • 12/07/2021

iostream
istream
wiostream
wistream

iostream

A type `basic_iostream` specialized on `char`.

```
C++
```

```
typedef basic_iostream<char, char_traits<char>> iostream;
```

Remarks

The type is a synonym for class template basic_iostream, specialized for elements of type `char` with default character traits.

istream

A type `basic_istream` specialized on `char`.

```
C++
```

```
typedef basic_istream<char, char_traits<char>> istream;
```

Remarks

The type is a synonym for class template basic_istream, specialized for elements of type `char` with default character traits.

wiostream

A type `basic_iostream` specialized on `wchar_t`.

```
```

```
C++
```

```
typedef basic_iostream<wchar_t, char_traits<wchar_t>> wiostream;
```

Remarks

The type is a synonym for class template basic_iostream, specialized for elements of type `wchar_t` with default character traits.

wistream

A type `basic_istream` specialized on `wchar_t`.

```
C++
```

```
typedef basic_istream<wchar_t, char_traits<wchar_t>> wistream;
```

Remarks

The type is a synonym for class template basic_istream, specialized for elements of type `wchar_t` with default character traits.

See also

<istream>

basic_iostream Class

Article • 08/03/2021

A stream class that can do both input and output.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_iostream : public basic_istream<Elem, Tr>,
    public basic_ostream<Elem, Tr>
{
public:
    explicit basic_iostream(basic_streambuf<Elem, Tr>* strbuf);

    virtual ~basic_iostream();

};
```

Remarks

The class template describes an object that controls insertions, through its base class basic_ostream< `Elem`, `Tr`>, and extractions, through its base class basic_istream< `Elem`, `Tr`>. The two objects share a common virtual base class basic_ios< `Elem`, `Tr`>. They also manage a common stream buffer, with elements of type `Elem`, whose character traits are determined by the class `Tr`. The constructor initializes its base classes through `basic_istream`(**strbuf**) and `basic_ostream`(**strbuf**).

Constructors

Constructor	Description
basic_iostream	Create a `basic_iostream` object.

Member functions

Member function	Description

Member function	Description
swap	Exchanges the contents of the provided `basic_iostream` object for the contents of this object.

Operators

Operator	Description
operator=	Assigns the value of a specified `basic_iostream` object to this object. This is a move assignment involving an `rvalue` that does not leave a copy behind.

Requirements

Header: <istream>

Namespace: std

basic_iostream::basic_iostream

Create a `basic_iostream` object.

```
C++

explicit basic_iostream(basic_streambuf<Elem, Tr>* strbuf);

basic_iostream(basic_iostream&& right);

basic_iostream();
```

Parameters

strbuf
An existing `basic_streambuf` object.

right
An existing `basic_iostream` object that is used to construct a new `basic_iostream`.

Remarks

The first constructor initializes the base objects by way of `basic_istream(strbuf)` and `basic_ostream(strbuf)`.

The second constructor initializes the base objects by calling `move(right)`.

basic_iostream::operator=

Assign the value of a specified `basic_iostream` object to this object. This is a move assignment involving an rvalue that does not leave a copy behind.

```
C++
```

```
basic_iostream& operator=(basic_iostream&& right);
```

Parameters

right
An `rvalue` reference to a `basic_iostream` object to assign from.

Remarks

The member operator calls `swap(right)`.

basic_iostream::swap

Exchanges the contents of the provided `basic_iostream` object for the contents of this object.

```
C++
```

```
void swap(basic_iostream& right);
```

Parameters

right
The `basic_iostream` object to swap.

Remarks

The member function calls `swap(right)`.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`basic_istream` Class

Article • 10/14/2022

Describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type `Char_T`, also known as char_type, whose character traits are determined by the class `Tr`, also known as traits_type.

Syntax

C++

```
template <class Char_T, class Tr = char_traits<Char_T>>
class basic_istream : virtual public basic_ios<Char_T, Tr>
```

Remarks

Most of the member functions that overload operator>> are formatted input functions. They follow the pattern:

C++

```
iostate state = goodbit;
const sentry ok(*this);

if (ok)
{
    try
    {
        /*extract elements and convert
            accumulate flags in state.
            store a successful conversion*/
    }
    catch (...)
    {
        try
        {
            setstate(badbit);

        }
        catch (...)
        {
        }
        if ((exceptions()& badbit) != 0)
            throw;
    }
```

```
}
setstate(state);

return (*this);
```

Many other member functions are unformatted input functions. They follow the pattern:

```
C++

iostate state = goodbit;
count = 0;    // the value returned by gcount
const sentry ok(*this, true);

if (ok)
{
    try
    {
        /* extract elements and deliver
            count extracted elements in count
            accumulate flags in state */
    }
    catch (...)
    {
        try
        {
            setstate(badbit);

        }
        catch (...)
        {
        }
        if ((exceptions()& badbit) != 0)
            throw;
    }
}
setstate(state);
```

Both groups of functions call `setstate(eofbit)` if they encounter end of file while extracting elements. For more information, see setstate.

An object of class `basic_istream<Char_T, Tr>` stores:

- A virtual public base object of class `basic_ios<Char_T, Tr>`. For more information, see basic_ios.
- An extraction count for the last unformatted input operation (called `count` in the previous code).

Example

See the example for basic_ifstream Class to learn more about input streams.

Constructors

Constructor	Description
basic_istream	Constructs an object of type `basic_istream`.

Member functions

Member function	Description
gcount	Returns the number of characters read during the last unformatted input.
get	Reads one or more characters from the input stream.
getline	Reads a line from the input stream.
ignore	Causes a number of elements to be skipped from the current read position.
peek	Returns the next character to be read.
putback	Puts a specified character into the stream.
read	Reads a specified number of characters from the stream and stores them in an array.
readsome	Read from buffer only.
seekg	Moves the read position in a stream.
sentry	The nested class describes an object whose declaration structures the formatted input functions and the unformatted input functions.
swap	Exchanges this `basic_istream` object for the provided `basic_istream` object parameter.
sync	Synchronizes the stream's associated input device with the stream's buffer.
tellg	Reports the current read position in the stream.
unget	Puts the most recently read character back into the stream.

Operators

Operator	Description
operator>>	Calls a function on the input stream or reads formatted data from the input stream.
operator=	Assigns the `basic_istream` on the right side of the operator to this object. It's a move assignment involving an `rvalue` reference that doesn't leave a copy behind.

Requirements

Header: `<istream>`

Namespace: `std`

`basic_istream::basic_istream`

Constructs an object of type `basic_istream`.

```
C++
```

```
explicit basic_istream(
    basic_streambuf<Char_T, Tr>* strbuf,
    bool _Isstd = false);

basic_istream(basic_istream&& right);
```

Parameters

`strbuf`
An object of type basic_streambuf.

`_Isstd`
`true` if it's a standard stream; otherwise, **`false`**.

`right`
A `basic_istream` object to copy.

Remarks

The first constructor initializes the base class by calling `init(strbuf)`. It also stores zero in the extraction count. For more information, see init. And for more information about this extraction count, see the Remarks section of the basic_istream Class overview.

The second constructor initializes the base class by calling `move(right)`. It also stores `right.gcount()` in the extraction count and stores zero in the extraction count for *`right`*.

Example

See the example for basic_ifstream::basic_ifstream to learn more about input streams.

`basic_istream::gcount`

Returns the number of characters read during the last unformatted input.

C++

```
streamsize gcount() const;
```

Return Value

The extraction count.

Remarks

Use basic_istream::get to read unformatted characters.

Example

C++

```
// basic_istream_gcount.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   cout << "Type the letter 'a': ";

   ws( cin );
   char c[10];

   cin.get( &c[0],9 );
   cout << c << endl;
```

```
    cout << cin.gcount( ) << endl;
}
```

Input

```
a
```

Output

```
Type the letter 'a': a
1
```

`basic_istream::get`

Reads one or more characters from the input stream.

C++

```
int_type get();

basic_istream<Char_T, Tr>& get(Char_T& Ch);
basic_istream<Char_T, Tr>& get(Char_T* str, streamsize count);
basic_istream<Char_T, Tr>& get(Char_T* str, streamsize count, Char_T
delimiter);

basic_istream<Char_T, Tr>& get(basic_streambuf<Char_T, Tr>& strbuf);
basic_istream<Char_T, Tr>& get(basic_streambuf<Char_T, Tr>& strbuf, Char_T
delimiter);
```

Parameters

`count`

The number of characters to read from *`strbuf`*.

`delimiter`

The character that should terminate the read if it's encountered before *`count`*.

`str`

A string in which to write.

`Ch`

A character to get.

strbuf

A buffer in which to write.

Return Value

The parameterless form of `get` returns the element read as an integer or end of file. The remaining forms return the stream (`*this`).

Remarks

The first unformatted input function extracts an element, if possible, as if by returning `rdbuf->sbumpc`. Otherwise, it returns `traits_type::`eof. If the function extracts no element, it calls `setstate(failbit)`. For more information, see setstate.

The second function extracts the int_type element `meta` the same way. If `meta` compares equal to `traits_type::eof`, the function calls `setstate(failbit)`. Otherwise, it stores `traits_type::to_char_type(meta)` in *`Ch`*. The function returns `*this`. For more information, see to_char_type.

The third function returns `get(str, count, widen('\n'))`.

The fourth function extracts up to `count - 1` elements and stores them in the array beginning at *`str`*. It always stores `char_type` after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *`delimiter`*. In this case, the element is put back to the controlled sequence.
- After the function extracts `count - 1` elements.

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

The fifth function returns `get(strbuf, widen('\n'))`.

The sixth function extracts elements and inserts them in *`strbuf`*. Extraction stops on end-of-file or on an element that compares equal to *`delimiter`*, which isn't extracted. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

Example

C++

```
// basic_istream_get.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10];

   c[0] = cin.get( );
   cin.get( c[1] );
   cin.get( &c[2],3 );
   cin.get( &c[4], 4, '7' );

   cout << c << endl;
}
```

Output

```
1111
```

basic_istream::getline

Gets a line from the input stream.

C++

```
basic_istream<Char_T, Tr>& getline(
    char_type* str,
    streamsize count);

basic_istream<Char_T, Tr>& getline(
    char_type* str,
    streamsize count,
    char_type delimiter);
```

Parameters

count

The number of characters to read from *strbuf*.

delimiter

The character that should terminate the read if it's encountered before *count*.

str

A string in which to write.

Return Value

The stream (*this).

Remarks

The first of these unformatted input functions returns getline(str, count, widen('\n')).

The second function extracts up to count - 1 elements and stores them in the array beginning at *str*. It always stores the string termination character after any extracted elements it stores. In order of testing, extraction stops:

- At end of file.
- After the function extracts an element that compares equal to *delimiter*. In this case, the element isn't put back, and it's not appended to the controlled sequence.
- After the function extracts count - 1 elements.

If the function extracts no elements or count - 1 elements, it calls setstate(failbit). In any case, it returns *this. For more information, see setstate.

Example

C++

```
// basic_istream_getline.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10];

   cin.getline( &c[0], 5, '2' );
   cout << c << endl;
}
```

Output

```
121
```

`basic_istream::ignore`

Causes a number of elements to be skipped from the current read position.

C++

```
basic_istream<Char_T, Tr>& ignore(
    streamsize count = 1,
    int_type delimiter = traits_type::eof());
```

Parameters

`count`

The number of elements to skip from the current read position.

`delimiter`

The element that, if encountered before count, causes `ignore` to return and allowing all elements after *`delimiter`* to be read.

Return Value

The stream (`*this`).

Remarks

The unformatted input function extracts up to *`count`* elements and discards them. If *`count`* equals `numeric_limits<int>::max`, however, it's taken as arbitrarily large. Extraction stops early on end of file or on an element `Ch` such that `traits_type::to_int_type(Ch)` compares equal to *`delimiter`* (which is also extracted). The function returns `*this`. For more information, see to_int_type.

Example

C++

```cpp
// basic_istream_ignore.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
   using namespace std;
   char chararray[10];
   cout << "Type 'abcdef': ";
   cin.ignore( 5, 'c' );
   cin >> chararray;
   cout << chararray;
}
```

Output

```
Type 'abcdef': abcdef
def
```

basic_istream::operator>>

Calls a function on the input stream or reads formatted data from the input stream.

C++

```cpp
basic_istream& operator>>(basic_istream& (* Pfn)(basic_istream&));
basic_istream& operator>>(ios_base& (* Pfn)(ios_base&));
basic_istream& operator>>(basic_ios<Char_T, Tr>& (* Pfn)(basic_ios<Char_T,
Tr>&));
basic_istream& operator>>(basic_streambuf<Char_T, Tr>* strbuf);
basic_istream& operator>>(bool& val);
basic_istream& operator>>(short& val);
basic_istream& operator>>(unsigned short& val);
basic_istream& operator>>(int& val);
basic_istream& operator>>(unsigned int& val);
basic_istream& operator>>(long& val);
basic_istream& operator>>(unsigned long& val);
basic_istream& operator>>(long long& val);
basic_istream& operator>>(unsigned long long& val);
basic_istream& operator>>(void *& val);
basic_istream& operator>>(float& val);
basic_istream& operator>>(double& val);
basic_istream& operator>>(long double& val);
```

Parameters

Pfn

A function pointer.

strbuf

An object of type `stream_buf`.

val

The value to read from the stream.

Return Value

The stream (`*this`).

Remarks

The `<istream>` header also defines several global extraction operators. For more information, see operator>> (\<istream>).

The first member function ensures that an expression of the form `istr >> ws` calls `ws(istr)`, and then returns `*this`. For more information, see ws.

The second and third functions ensure that other manipulators, such as hex, behave similarly. The remaining functions are the formatted input functions.

The function:

```
C++

basic_istream& operator>>(
    basic_streambuf<Char_T, Tr>* strbuf);
```

extracts elements, if *strbuf* isn't a null pointer, and inserts them in *strbuf*. Extraction stops on end of file. It also stops without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`. For more information, see setstate.

The function:

```
C++

basic_istream& operator>>(bool& val);
```

extracts a field and converts it to a Boolean value by calling `use_facet< num_get<Char_T, InIt>(getloc).get( InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator<Char_T, Tr>`. The function returns `*this`.

For more information, see use_facet, getloc, get, rdbuf, and istreambuf_iterator.

Each of the functions:

```
C++

basic_istream& operator>>(short& val);
basic_istream& operator>>(unsigned short& val);
basic_istream& operator>>(int& val);
basic_istream& operator>>(unsigned int& val);
basic_istream& operator>>(long& val);
basic_istream& operator>>(unsigned long& val);
basic_istream& operator>>(long long& val);
basic_istream& operator>>(unsigned long long& val);
basic_istream& operator>>(void *& val);
```

extract a field and convert it to a numeric value by calling `use_facet<num_get<Char_T, InIt>(getloc).get(InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator<Char_T, Tr>`, and *`val`* has type **`long`**, **`unsigned long`**, or **`void *`** as needed.

If the converted value can't be represented as the type of *`val`*, the function calls `setstate(failbit)`. In any case, the function returns `*this`. For more information, see setstate.

Each of the functions:

```
C++

basic_istream& operator>>(float& val);
basic_istream& operator>>(double& val);
basic_istream& operator>>(long double& val);
```

extract a field and convert it to a numeric value by calling `use_facet<num_get<Char_T, InIt>(getloc).get(InIt(rdbuf), Init(0), *this, getloc, val)`. Here, `InIt` is defined as `istreambuf_iterator<Char_T, Tr>`, and *`val`* has type **`double`** or **`long double`** as needed.

If the converted value can't be represented as the type of *`val`*, the function calls `setstate(failbit)`. In any case, it returns `*this`.

Example

C++

```
// istream_basic_istream_op_is.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

ios_base& hex2( ios_base& ib )
{
   ib.unsetf( ios_base::dec );
   ib.setf( ios_base::hex );
   return ib;
}

basic_istream<char, char_traits<char> >& somefunc(basic_istream<char,
char_traits<char> > &i)
{
   if ( i == cin )
   {
      cerr << "i is cin" << endl;
   }
   return i;
}

int main( )
{
   int i = 0;
   cin >> somefunc;
   cin >> i;
   cout << i << endl;
   cin >> hex2;
   cin >> i;
   cout << i << endl;
}
```

`basic_istream::operator=`

Assigns the `basic_istream` on the right side of the operator to this object. It's a move assignment involving an `rvalue` reference that doesn't leave a copy behind.

C++

```
basic_istream& operator=(basic_istream&& right);
```

Parameters

right

An `rvalue` reference to a `basic_ifstream` object.

Return Value

Returns `*this`.

Remarks

The member operator calls `swap(right)`.

`basic_istream::peek`

Returns the next character to be read.

```
C++

int_type peek();
```

Return Value

The next character that will be read.

Remarks

The unformatted input function extracts an element, if possible, as if by returning `rdbuf->sgetc`. Otherwise, it returns `traits_type::eof`. For more information, see sgetc and eof.

Example

```
C++

// basic_istream_peek.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2;
   cout << "Type 'abcde': ";
```

```
    c2 = cin.peek( );
    cin.getline( &c[0], 9 );

    cout << c2 << " " << c << endl;
}
```

Input

```
abcde
```

Output

```
Type 'abcde': abcde
a abcde
```

`basic_istream::putback`

Puts a specified character into the stream.

C++

```
basic_istream<Char_T, Tr>& putback(
    char_type Ch);
```

Parameters

`Ch`

A character to put back into the stream.

Return Value

The stream (`*this`).

Remarks

The unformatted input function puts back *`Ch`*, if possible, as if by calling `rdbuf->sputbackc`. If `rdbuf` is a null pointer, or if the call to `sputbackc` returns `traits_type::eof`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

For more information, see rdbuf, sputbackc, eof, and setstate.

Example

C++

```
// basic_istream_putback.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2, c3;

   c2 = cin.get( );
   c3 = cin.get( );
   cin.putback( c2 );
   cin.getline( &c[0], 9 );
   cout << c << endl;
}
```

Output

```
qwq
```

`basic_istream::read`

Reads a specified number of characters from the stream and stores them in an array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

C++

```
basic_istream<Char_T, Tr>& read(
    char_type* str,
    streamsize count);
```

Parameters

`str`

The array in which to read the characters.

`count`

The number of characters to read.

Return Value

The stream (`*this`).

Remarks

The unformatted input function extracts up to `count` elements and stores them in the array beginning at `str`. Extraction stops early on end of file, in which case the function calls `setstate(failbit)`. In any case, it returns `*this`. For more information, see setstate.

Example

C++

```
// basic_istream_read.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main()
{
    char c[10];
    int count = 5;

    cout << "Type 'abcde': ";

    // Note: cin::read is potentially unsafe, consider
    // using cin::_Read_s instead.
    cin.read(&c[0], count);
    c[count] = 0;

    cout << c << endl;
}
```

Input

```
abcde
```

Output

```
Type 'abcde': abcde
abcde
```

`basic_istream::readsome`

Reads the specified number of character values.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
C++
```

```
streamsize readsome(
    char_type* str,
    streamsize count);
```

Parameters

`str`
The array in which `readsome` stores the characters it reads.

`count`
The number of characters to read.

Return Value

The number of characters actually read, gcount.

Remarks

This unformatted input function extracts up to *`count`* elements from the input stream and stores them in the array *`str`*.

This function doesn't wait for input. It reads whatever data is available.

Example

```
C++
```

```
// basic_istream_readsome.cpp
// compile with: /EHsc /W3
#include <iostream>
using namespace std;

int main( )
{
```

```
    char c[10];
    int count = 5;

    cout << "Type 'abcdefgh': ";

    // cin.read blocks until user types input.
    // Note: cin::read is potentially unsafe, consider
    // using cin::_Read_s instead.
    cin.read(&c[0], 2);

    // Note: cin::readsome is potentially unsafe, consider
    // using cin::_Readsome_s instead.
    int n = cin.readsome(&c[0], count);  // C4996
    c[n] = 0;
    cout << n << " characters read" << endl;
    cout << c << endl;
}
```

`basic_istream::seekg`

Moves the read position in a stream.

C++

```
basic_istream<Char_T, Tr>& seekg(pos_type pos);

basic_istream<Char_T, Tr>& seekg(off_type off, ios_base::seekdir way);
```

Parameters

`pos`

The absolute position in which to move the read pointer.

`off`

An offset to move the read pointer relative to *`way`*.

`way`

One of the ios_base::seekdir enumerations.

Return Value

The stream (`*this`).

Remarks

The first member function performs an absolute seek, the second member function performs a relative seek.

> ⓘ Note
>
> Do not use the second member function with text files, because Standard C++ does not support relative seeks in text files.

If fail is `false`, the first member function calls `newpos = rdbuf->pubseekpos(pos)`, for some `pos_type` temporary object `newpos`. If `fail` is `false`, the second function calls `newpos = rdbuf->pubseekoff( off, way)`. In either case, if `(off_type)newpos == (off_type)(-1)` (the positioning operation fails), the function calls `istr.setstate(failbit)`. Both functions return `*this`.

If fail is `true`, the member functions do nothing.

For more information, see rdbuf, pubseekpos, pubseekoff, and setstate.

Example

C++

```
// basic_istream_seekg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main ( )
{
   using namespace std;
   ifstream file;
   char c, c1;

   file.open( "basic_istream_seekg.txt" );
   file.seekg(2);   // seek to position 2
   file >> c;
   cout << c << endl;
}
```

`basic_istream::sentry`

The nested class describes an object whose declaration structures the formatted and unformatted input functions.

```
C++
```

```
class sentry {
   public:
   explicit sentry(
      basic_istream<Char_T, Tr>& _Istr,
      bool _Noskip = false);
   operator bool() const;
   };
```

Remarks

If `_Istr.good` is `true`, the constructor:

- Calls `_Istr.tie->flush` if `_Istr.tie` isn't a null pointer.
- Effectively calls `ws(_Istr)` if `_Istr.flags & skipws` is nonzero.

If after any such preparation, `_Istr.good` is `false`, the constructor calls `_Istr.setstate(failbit)`. In any case, the constructor stores the value returned by `_Istr.good` in `status`. A later call to `operator bool` delivers this stored value.

For more information, see good, tie, flush, ws, flags, skipws, and setstate.

`basic_istream::swap`

Exchanges the contents of two `basic_istream` objects.

```
C++
```

```
void swap(basic_istream& right);
```

Parameters

`right`
An `lvalue` reference to a `basic_istream` object.

Remarks

The member function calls `basic_ios::swap(right)`. It also exchanges the extraction count with the extraction count for *`right`*. For more information, see basic_ios::swap.

`basic_istream::sync`

Synchronizes the stream's associated input device with the stream's buffer.

```
C++
```

```
int sync();
```

Return Value

If rdbuf is a null pointer, the function returns -1. Otherwise, it calls `rdbuf->pubsync`. If that call returns -1, the function calls `setstate(badbit)` and returns -1. Otherwise, the function returns zero. For more information, see pubsync and setstate.

`basic_istream::tellg`

Reports the current read position in the stream.

```
C++
```

```
pos_type tellg();
```

Return Value

The current position in the stream.

Remarks

If fail is `false`, the member function returns `rdbuf->pubseekoff(0, cur, in)`. Otherwise, it returns `pos_type(-1)`. For more information, see rdbuf and pubseekoff.

Example

```
C++
```

```
// basic_istream_tellg.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
```

```
{
    using namespace std;
    ifstream file;
    char c;
    streamoff i;

    file.open("basic_istream_tellg.txt");
    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;

    i = file.tellg();
    file >> c;
    cout << c << " " << i << endl;
}
```

`basic_istream::unget`

Puts the most recently read character back into the stream.

C++

```
basic_istream<Char_T, Tr>& unget();
```

Return Value

The stream (`*this`).

Remarks

The unformatted input function puts back the previous element in the stream, if possible, as if by calling `rdbuf->sungetc` If rdbuf is a null pointer, or if the call to `sungetc` returns `traits_type::eof`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

For more information, see sungetc, eof, and setstate. And for information on how `unget` might fail, see basic_streambuf::sungetc.

Example

C++

```
// basic_istream_unget.cpp
// compile with: /EHsc
```

```
#include <iostream>
using namespace std;

int main( )
{
   char c[10], c2;

   cout << "Type 'abc': ";
   c2 = cin.get( );
   cin.unget( );
   cin.getline( &c[0], 9 );
   cout << c << endl;
}
```

Input

```
abc
```

Output

```
Type 'abc': abc
abc
```

See also

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

`<iterator>`

Article • 12/21/2022

Defines predefined iterators and stream iterators, iterator primitives, and supporting templates.

Requirements

Header: `<iterator>`

Namespace: `std`

Remarks

Iterators are a generalization of pointers that allow a C++ program to work with different data structures in a uniform way. Instead of operating on specific data types, algorithms operate on a range of values as specified by a kind of iterator. Algorithms can operate on any data structure that satisfies the requirements of the iterator.

In C++20, there are six categories of iterators. Iterators are arranged in a hierarchy of capability. Their capabilities are specified by C++20 concepts. For a description of the various iterators and their capabilities, see Iterator concepts

Visual Studio has added extensions to C++ Standard Library iterators to support debugging for checked and unchecked iterators. For more information, see Safe Libraries: C++ Standard Library.

Members

Functions

Name	Description
advance	Increments an iterator by a specified number of positions.
back_inserter	Creates an iterator that can insert elements at the back of a specified container.
begin	Retrieves an iterator to the first element in a specified container.

Name	Description
cbegin	Retrieves a read-only iterator to the first element in a specified container.
cend	Retrieves a read-only iterator to the element that follows the last element in the specified container.
crbegin	Get a reverse read-only iterator to the beginning of the specified container.
crend	Get the sentinel at the end of what `crbegin()` returns.
data	Get a pointer to the first element in the specified container.
distance	Determines the number of increments between the positions addressed by two iterators.
end	Retrieves an iterator to the element that follows the last element in the specified container.
empty	Test if the specified container is empty.
front_inserter	Creates an iterator that can insert elements at the front of a specified container.
inserter	An iterator adaptor that adds a new element to a container at a specified point of insertion.
make_checked_array_iterator	Creates a checked_array_iterator that can be used by other algorithms. **Note:** This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function isn't portable to C++ Standard build environments that don't support this Microsoft extension.
make_move_iterator	Returns a move iterator containing the provided iterator as its stored base iterator.
make_unchecked_array_iterator	Creates an unchecked_array_iterator that can be used by other algorithms. **Note:** This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function isn't portable to C++ Standard build environments that don't support this Microsoft extension.
next	Iterates a specified number of times and returns the new iterator position.
prev	Iterates in reverse a specified number of times and returns the new iterator position.
rbegin	Get a reverse iterator to the beginning of the specified container.

Name	Description
rend	Get a reverse iterator to the sentinel at the end of the specified container.
size	Get the number of elements.

Operators

Name	Description
operator!=	Tests if the iterator object on the left side of the operator isn't equal to the iterator object on the right side.
operator==	Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.
operator<	Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.
operator<=	Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.
operator>	Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.
operator>=	Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.
operator+	Adds an offset to an iterator and returns the new `reverse_iterator` addressing the inserted element at the new offset position.
operator-	Subtracts one iterator from another and returns the difference.

Classes

Name	Description
back_insert_iterator	The class template describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container.
bidirectional_iterator_tag	A class that provides a return type for an `iterator_category` function that represents a bidirectional iterator.

Name	Description
checked_array_iterator	A class that accesses an array using a random access, checked iterator. **Note:** This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function isn't portable to C++ Standard build environments that don't support this Microsoft extension.
forward_iterator_tag	A class that provides a return type for an `iterator_category` function that represents a forward iterator.
front_insert_iterator	The class template describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container.
input_iterator_tag	A class that provides a return type for an `iterator_category` function that represents an input iterator.
insert_iterator	The class template describes an output iterator object. It inserts elements into a container of type `Container`, which it accesses through the protected `pointer` object it stores called container. It also stores the protected `iterator` object, of class `Container::iterator`, called `iter`.
istream_iterator	The class template describes an input iterator object. It extracts objects of class `Ty` from an input stream, which it accesses through an object it stores, of type pointer to `basic_istream<Elem, Tr>`.
istreambuf_iterator	The class template describes an input iterator object. It inserts elements of class `Elem` into an output stream buffer, which it accesses through an object it stores, of type `pointer` to `basic_streambuf<Elem, Tr>`.
iterator	The class template is used as a base type for all iterators.
iterator_traits	A template helper class providing critical types that are associated with different iterator types so that they can be referred to in the same way.
move_iterator	A `move_iterator` object stores a random-access iterator of type `RandomIterator`. It behaves like a random-access iterator, except when dereferenced. The result of `operator*` is implicitly cast to `value_type&&:` to make an `rvalue reference`.
ostream_iterator	The class template describes an output iterator object. It inserts objects of class `Type` into an output stream, which it accesses through an object it stores, of type `pointer` to `basic_ostream<Elem, Tr>`.

Name	Description
ostreambuf_iterator	The class template describes an output iterator object. It inserts elements of class `Elem` into an output stream buffer, which it accesses through an object it stores, of type pointer to `basic_streambuf<Elem, Tr>`.
output_iterator_tag	A class that provides a return type for `iterator_category` function that represents an output iterator.
random_access_iterator_tag	A class that provides a return type for `iterator_category` function that represents a random-access iterator.
reverse_iterator	The class template describes an object that behaves like a random-access iterator, only in reverse.
unchecked_array_iterator	A class that accesses an array using a random access, unchecked iterator. **Note:** This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function isn't portable to C++ Standard build environments that don't support this Microsoft extension.

Concepts

The following concepts are defined in the `std` namespace. They apply to iterators, and are also related to the iterator categories for ranges described in <ranges> concepts.

Iterator concept	Description
bidirectional_iterator C++20	Specifies an iterator that can read and write both forwards and backwards.
contiguous_iterator C++20	Specifies an iterator whose elements are sequential in memory, the same size, and can be accessed using pointer arithmetic.
forward_iterator C++20	Specifies an iterator that can read (and possibly write) multiple times.
input_iterator C++20	Specifies an iterator that you can read from at least once.
input_or_output_iterator C++20	The basis of the iterator concept taxonomy.
output_iterator	Specifies an iterator that you can write to.
random_access_iterator C++20	Specifies an iterator that you can read and write by index.
sentinel_for C++20	Specifies a sentinel for an iterator type.

Iterator concept	Description
sized_sentinel_for[C++20]	Specifies that an iterator and its sentinel can be subtracted (using `-`) to find their difference in constant time.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

Iterator concepts

Article • 12/21/2022

Concepts are a C++20 language feature that constrain template parameters at compile time. They help prevent incorrect template instantiation, specify template argument requirements in a readable form, and provide more succinct template related compiler errors.

Consider the following example, which defines a concept to prevent instantiating the template with a type that doesn't support division:

C++

```
// requires /std:c++20 or later
#include <iostream>

// Definition of dividable concept which requires
// that arguments a & b of type T support division
template <typename T>
concept dividable = requires (T a, T b)
{
    a / b;
};

// Apply the concept to a template.
// The template will only be instantiated if argument T supports division.
// This prevents the template from being instantiated with types that don't
support division.
// This could have been applied to the parameter of a template function, but
because
// most of the concepts in the <ranges> library are applied to classes, this
form is demonstrated.
template <class T> requires dividable<T>
class DivideEmUp
{
public:
    T Divide(T x, T y)
    {
        return x / y;
    }
};

int main()
{
    DivideEmUp<int> dividerOfInts;
    std::cout << dividerOfInts.Divide(6, 3); // outputs 2
    // The following line will not compile because the template can't be
instantiated
    // with char* because char* can be divided
    DivideEmUp<char*> dividerOfCharPtrs; // compiler error: cannot deduce
```

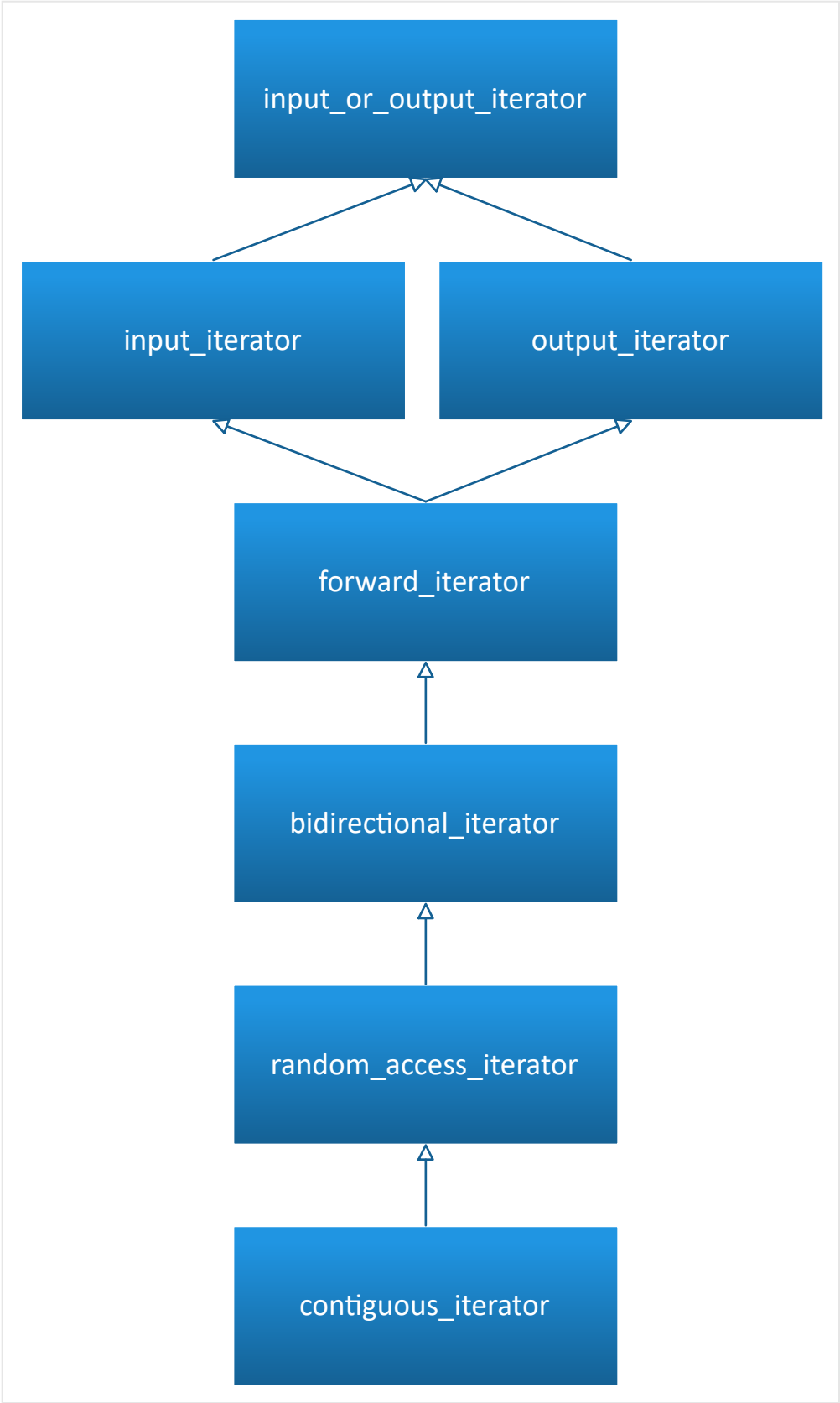
```
template arguments
}
```

When you pass the compiler switch `/diagnostics:caret` to Visual Studio 2022 version 17.4 preview 4 or later, the error that concept `dividable<char*>` evaluated to false will point directly to the expression requirement `(a / b)` that failed.

Iterator concepts are defined in the `std` namespace, and are declared in the `<iterator>` header file. They're used in the declarations of range adaptors, views, and so on.

There are six categories of iterators. They're directly related to the categories of ranges listed under Range concepts.

The following iterator concepts are listed in order of increasing capability. `input_or_output_iterator` is at the low end of the capability hierarchy, and `contiguous_iterator` is at the high end. Iterators higher in the hierarchy can generally be used in place of those that are lower, but not vice-versa. For example, a `random_access_iterator` iterator can be used in place of a `forward_iterator`, but not the other way around. An exception is `input_iterator`, which can't be used in place of `output_iterator` because it can't write.



In the following table, "Multi-pass" refers to whether the iterator can revisit the same element more than once. For example, `vector::iterator` is a multi-pass iterator because you can make a copy of the iterator, read the elements in the collection, and then restore the iterator to the value in the copy, and revisit the same elements again. If an iterator is single-pass, you can only visit the elements in the collection once.

In the following table, "Example types" refers to collections/iterators that satisfy the concept.

Iterator concept	Description	Direction	Read/write	Multi-pass	Example types
input_or_output_iterator C++20	The basis of the iterator concept taxonomy.	Forward	Read/write	no	`istream_iterator`, `ostream_iterator`
output_iterator C++20	Specifies an iterator that you can write to.	Forward	Write	no	`ostream`, `inserter`
input_iterator C++20	Specifies an iterator that you can read from once.	Forward	Read	no	`istream`, `istreambuf_iterator`
forward_iterator C++20	Specifies an iterator that can read (and possibly write) multiple times.	Forward	Read/write	yes	`vector`, `list`
bidirectional_iterator C++20	Specifies an iterator that you can read and write both forwards and backwards.	Forward or backward	Read/write	yes	`list`, `set`, `multiset`, `map`, and `multimap`.
random_access_iterator C++20	Specifies an iterator that you can read and write by index.	Forward or backward	Read/write	yes	`vector`, `array`, `deque`

Iterator concept	Description	Direction	Read/write	Multi-pass	Example types
contiguous_iterator[C++20]	Specifies an iterator whose elements are sequential in memory, are the same size, and can be accessed using pointer arithmetic.	Forward or backward	Read/write	yes	`array`, `vector` `string`.

Other iterator concepts include:

Iterator concept	Description
sentinel_for[C++20]	Specifies that a type is a sentinel for an iterator type.
sized_sentinel_for[C++20]	Specifies that an iterator and its sentinel can be subtracted (using `-`) to find their difference in constant time.

`bidirectional_iterator`

A `bidirectional_iterator` supports reading and writing forwards and backwards.

C++

```
template<class I>
concept bidirectional_iterator =
    forward_iterator<I> &&
    derived_from<ITER_CONCEPT(I), bidirectional_iterator_tag> &&
    requires(I i) {
        {--i} -> same_as<I&>;
        {i--} -> same_as<I>;
};
```

Parameters

I

The iterator to test to see if it's a `bidirectional_iterator`.

Remarks

A `bidirectional_iterator` has the capabilities of a `forward_iterator`, but can also iterate backwards.

Some examples of containers that can be used with a `bidirectional_iterator` are `set`, `multiset`, `map`, `multimap`, `vector`, and `list`.

Example: `bidirectional_iterator`

The following example uses the `bidirectional_iterator` concept to show that `vector<int>` has a `bidirectional_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    std::cout << std::boolalpha <<
std::bidirectional_iterator<std::vector<int>::iterator> << '\n'; // outputs
"true"

    // another way to test
    std::vector<int> v = {0,1,2};
    std::cout << std::boolalpha <<
std::contiguous_iterator<decltype(v)::iterator>; // outputs true
}
```

`contiguous_iterator`

Specifies an iterator whose elements are sequential in memory, are the same size, and can be accessed using pointer arithmetic.

C++

```
template<class I>
    concept contiguous_iterator =
        random_access_iterator<I> &&
        derived_from<ITER_CONCEPT(I), contiguous_iterator_tag> &&
        is_lvalue_reference_v<iter_reference_t<I>> &&
        same_as<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>>> &&
        requires(const I& i) {
            { to_address(i) } ->
```

```
same_as<add_pointer_t<iter_reference_t<I>>>;
        };
```

Parameters

I

The type to test to see if it's a `contiguous_iterator`.

Remarks

A `contiguous_iterator` can be accessed by pointer arithmetic because the elements are laid out sequentially in memory and are the same size. Some examples of a `contiguous_iterator` are `array`, `vector`, and `string`.

Example: `contiguous_iterator`

The following example uses the `contiguous_iterator` concept to show that a `vector<int>` has a `contiguous_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    // Show that vector<int> has a contiguous_iterator
    std::cout << std::boolalpha <<
std::contiguous_iterator<std::vector<int>::iterator> << '\n'; // outputs
"true"

    // another way to test
    std::vector<int> v = {0,1,2};
    std::cout << std::boolalpha <<
std::contiguous_iterator<decltype(v)::iterator>; // outputs true
}
```

`forward_iterator`

Has the capabilities of an `input_iterator` and an `output_iterator`. Supports iterating over a collection multiple times.

```
```

C++

```
template<class I>
    concept forward_iterator =
        input_iterator<I> &&
        derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
        incrementable<I> &&
        sentinel_for<I, I>;
```

Parameters

I

The iterator to test to see if it's a `forward_iterator`.

Remarks

A `forward_iterator` can only move forward.

Some examples of containers that can be used with a `forward_iterator` are `vector`, `list`, `unordered_set`, `unordered_multiset`, `unordered_map`, and `unordered_multimap`.

Example: `forward_iterator`

The following example uses the `forward_iterator` concept to show that a `vector<int>` has a `forward_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    // Show that vector has a forward_iterator
    std::cout << std::boolalpha <<
std::forward_iterator<std::vector<int>::iterator> << '\n'; // outputs "true"

    // another way to test
    std::vector<int> v = {0,1,2};
    std::cout << std::boolalpha <<
std::forward_iterator<decltype(v)::iterator>; // outputs true
}
```

`input_iterator`

An `input_iterator` is an iterator that you can read from at least once.

```
C++
```

```
template<class I>
concept input_iterator =
    input_or_output_iterator<I> &&
    indirectly_readable<I> &&
    requires { typename ITER_CONCEPT(I); } &&
    derived_from<ITER_CONCEPT(I), input_iterator_tag>;
```

Parameters

`I`

The type to test to see if it's an `input_iterator`.

Remarks

Calling `begin()` on an `input_iterator` more than once results in undefined behavior. A type that only models `input_iterator` isn't multi-pass. Consider reading from standard input (`cin`) for example. In this case, you can only read the current element once and you can't re-read characters you've already read. An `input_iterator` only reads forward, not backwards.

Example: `input_iterator`

The following example uses the `input_iterator` concept to show that an `istream_iterator` has an `input_iterator`:

```
C++
```

```
// requires /std:c++20 or later
#include <iostream>

int main()
{
    // Show that a istream_iterator has an input_iterator
    std::cout << std::boolalpha <<
std::input_iterator<std::istream_iterator<int>>; // outputs true
}
```

`input_or_output_iterator`

An `input_or_output_iterator` is the basis of the iterator concept taxonomy. It supports dereferencing and incrementing an iterator. Every iterator models `input_or_output_iterator`.

C++

```
template<class I>
concept input_or_output_iterator =
    requires(I i) {
        { *i } -> can-reference;
    } &&
    weakly_incrementable<I>;
```

Parameters

I

The type to test to see if it's an `input_or_output_iterator`.

Remarks

The concept `can-reference` means that the type `I` is a reference, a pointer, or a type that can be implicitly converted to a reference.

Example: `input_or_output_iterator`

The following example uses the `input_or_output_iterator` concept to show that `vector<int>` has an `input_or_output_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>

int main()
{
    // Show that a vector has an input_or_output_iterator
    std::cout << std::boolalpha <<
std::input_or_output_iterator<std::vector<int>::iterator> << '\n'; //
outputs true

    // another way to test
    std::vector<int> v = {0,1,2};
    std::cout << std::boolalpha <<
```

```
std::input_or_output_iterator<decltype(v)::iterator>; // outputs true
}
```

`output_iterator`

An `output_iterator` is an iterator that you can write to.

C++

```
template<class I, class T>
concept output_iterator =
    input_or_output_iterator<I> &&
    indirectly_writable<I, T> &&
    requires(I i, T&& t) {
        *i++ = std::forward<T>(t);
    };
```

Parameters

I

The type to test to see if it's an `output_iterator`.

T

The type of the values to write.

Remarks

An `output_iterator` is single pass. That is, it can only write to the same element once.

Example: `output_iterator`

The following example uses the `output_iterator` concept to show that `vector<int>` has an `output_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    // Show that vector<int> has an output_iterator
    std::cout << std::boolalpha <<
```

```
std::output_iterator<std::vector<int>::iterator, int> << "\n"; // outputs
"true"

    // another way to test
    std::vector<int> v = {0,1,2,3,4,5};
    std::cout << std::boolalpha <<
std::output_iterator<decltype(v)::iterator, int>; // outputs true
}
```

`random_access_iterator`

A `random_access_iterator` can read or write by index.

C++

```
template<class I>
concept random_access_iterator =
    bidirectional_iterator<I> &&
    derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
    totally_ordered<I> &&
    sized_sentinel_for<I, I> &&
    requires(I i, const I j, const iter_difference_t<I> n) {
        { i += n } -> same_as<I&>;
        { j + n } -> same_as<I>;
        { n + j } -> same_as<I>;
        { i -= n } -> same_as<I&>;
        { j - n } -> same_as<I>;
        { j[n] } -> same_as<iter_reference_t<I>>;
    };
```

Parameters

I

The type to test to see if it's a `random_access_iterator`.

Remarks

A `random_access_iterator` has the capabilities of an `input_iterator`, `output_iterator`, `forward_iterator`, and `bidirectional_iterator`.

Some examples of a `random_access_iterator` are `vector`, `array`, and `deque`.

Example: `random_access_iterator`

The following example shows that a `vector<int>` has a `random_access_iterator`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    // Show that vector<int> has a random_access_iterator
    std::cout << std::boolalpha <<
std::random_access_iterator<std::vector<int>::iterator> << '\n'; // outputs
"true"

    // another way to test
    std::vector<int> v = {0,1,2};
    std::cout << std::boolalpha <<
std::random_access_iterator<decltype(v)::iterator>; // outputs true
}
```

`sentinel_for`

Specifies that a type is a sentinel for an iterator.

C++

```
template<class S, class I>
concept sentinel_for =
    semiregular<S> &&
    input_or_output_iterator<I> &&
    weakly-equality-comparable-with <S, I>;
```

Parameters

`I`

The iterator type.

`S`

The type to test to see if it's a sentinel for `I`.

Remarks

A sentinel is a type that can be compared to an iterator to determine if the iterator has reached the end. This concept determines if a type is a sentinel for one of the

`input_or_output_iterator` types, which includes `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, `random_access_iterator`, and `contiguous_iterator`.

Example: `sentinel_for`

The following example uses the `sentinel_for` concept to show that `vector<int>::iterator` is a sentinel for `vector<int>`:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2};
    std::vector<int>::iterator i = v.begin();
    // show that vector<int>::iterator is a sentinel for vector<int>
    std::cout << std::boolalpha <<
std::sentinel_for<std::vector<int>::iterator, decltype(i)>; // outputs true
}
```

`sized_sentinel_for`

Test that an iterator and its sentinel can be subtracted using `-` to find the difference, in constant time.

C++

```
template<class S, class I>
concept sized_sentinel_for =
    sentinel_for<S, I> &&
    !disable_sized_sentinel_for<remove_cv_t<S>, remove_cv_t<I>> &&
    requires(const I& i, const S& s) {
        {s - i} -> same_as<iter_difference_t<I>>;
        {i - s} -> same_as<iter_difference_t<I>>;
    };
```

Parameters

`I`

The iterator type.

`S`

The sentinel type to test.

Remarks

Example: `sized_sentinel_for`

The following example uses the `sized_sentinel_for` concept to verify that the sentinel for a `vector<int>` can be subtracted from the vectors iterator in constant time:

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = { 1, 2, 3 };
    std::vector<int>::iterator i = v.begin();
    std::vector<int>::iterator end = v.end();
    // use the sized_sentinel_for concept to verify that i can be subtracted
from end in constant time
    std::cout << std::boolalpha << std::sized_sentinel_for<decltype(end),
decltype(i)> << "\n"; // outputs true
    std::cout << end - i; // outputs 3
}
```

See also

Range concepts

Range adaptors

View classes

`<iterator>` functions

Article • 01/30/2023

`advance`

Increments an iterator by a specified number of positions.

C++

```
template <class InputIterator, class Distance>
void advance(InputIterator& InIt, Distance Off);
```

Parameters

`InIt`

The iterator that is to be incremented and that must satisfy the requirements for an input iterator.

`Off`

An integral type that is convertible to the iterator's difference type and that specifies the number of increments the position of the iterator is to be advanced.

Remarks

The range must be nonsingular, where the iterators must be dereferenceable or past the end.

If the `InputIterator` satisfies the requirements for a bidirectional iterator type, then *`Off`* may be negative. If `InputIterator` is an input or forward iterator type, *`Off`* must be nonnegative.

The advance function has constant complexity when `InputIterator` satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

C++

```cpp
// iterator_advance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main()
{
    using namespace std;

    list<int> L;
    for (int i = 1; i < 9; ++i)
    {
        L.push_back(i);
    }
    list<int>::iterator LPOS = L.begin();

    cout << "The list L is: ( ";
    for (auto L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;

    cout << "The iterator LPOS initially points to the first element: "
        << *LPOS << "." << endl;

    advance(LPOS, 4);
    cout << "LPOS is advanced 4 steps forward to point"
        << " to the fifth element: "
        << *LPOS << "." << endl;

    advance(LPOS, -3);
    cout << "LPOS is moved 3 steps back to point to the "
        << "2nd element: " << *LPOS << "." << endl;
}
```

Output

```
The list L is: ( 1 2 3 4 5 6 7 8 ).
The iterator LPOS initially points to the first element: 1.
LPOS is advanced 4 steps forward to point to the fifth element: 5.
LPOS is moved 3 steps back to point to the 2nd element: 2.
```

back_inserter

Creates an iterator that can insert elements at the back of a specified container.

C++

```
template <class Container>
back_insert_iterator<Container> back_inserter(Container& Cont);
```

Parameters

Cont

The container into which the back insertion is to be executed.

Return Value

A `back_insert_iterator` associated with the container object *Cont*.

Remarks

Within the C++ Standard Library, the argument must refer to one of the three sequence containers that have the member function `push_back`: deque Class, list Class, or vector Class.

Example

C++

```
// iterator_back_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 3; ++i)
    {
        vec.push_back(i);
    }

    cout << "The initial vector vec is: ( ";
    for (auto vIter = vec.begin(); vIter != vec.end(); vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    // Insertions can be done with template function
```

```
    back_insert_iterator<vector<int> > backiter(vec);
    *backiter = 30;
    backiter++;
    *backiter = 40;

    // Alternatively, insertions can be done with the
    // back_insert_iterator member function
    back_inserter(vec) = 500;
    back_inserter(vec) = 600;

    cout << "After the insertions, the vector vec is: ( ";
    for (auto vIter = vec.begin(); vIter != vec.end(); vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;
}
```

Output

```
The initial vector vec is: ( 0 1 2 ).
After the insertions, the vector vec is: ( 0 1 2 30 40 500 600 ).
```

`begin`

Retrieves an iterator to the first element in a specified container.

C++

```
template <class Container>
auto begin(Container& cont)  `
   -> decltype(cont.begin());

template <class Container>
auto begin(const Container& cont)   `
   -> decltype(cont.begin());

template <class Ty, class Size>
Ty *begin(Ty (& array)[Size]);
```

Parameters

cont

A container.

array

An array of objects of type `Ty`.

Return Value

The first two template functions return `cont.begin()`. The first function is non-constant; the second one is constant.

The third template function returns *array*.

Example

We recommend that you use this template function in place of container member `begin()` when more generic behavior is required.

C++

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

template <typename C> void reverse_sort(C& c)
{
    std::sort(std::begin(c), std::end(c), std::greater<>());
}

template <typename C> void print(const C& c)
{
    for (const auto& e : c)
    {
        std::cout << e << " ";
    }
    std::cout << "\n";
}

int main()
{
    std::vector<int> v = { 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4,
2, 1 };

    print(v);
    reverse_sort(v);
    print(v);

    std::cout << "--\n";
```

```
    int arr[] = { 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2,
1 };

    print(arr);
    reverse_sort(arr);
    print(arr);
}
```

Output

```
11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
52 40 34 26 20 17 16 13 11 10 8 5 4 2 1
--
23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
160 106 80 70 53 40 35 23 20 16 10 8 5 4 2 1
```

The function `reverse_sort` supports containers of any kind, in addition to regular arrays, because it calls the non-member version of `begin()`. Coding `reverse_sort` to use the container member `begin()`:

C++

```
template <typename C>
void reverse_sort(C& c) {
    using std::begin;
    using std::end;

    std::sort(c.begin(), c.end(), std::greater<>());

}
```

Then sending an array to it, causes this compiler error:

Output

```
error C2228: left of '.begin' must have class/struct/union
```

cbegin

Retrieves a const (read-only) iterator to the first element in a specified container.

C++

```
template <class Container>
auto cbegin(const Container& cont)
   -> decltype(cont.begin());
```

Parameters

cont

A container or `initializer_list`.

Return Value

A constant `cont.begin()`.

Remarks

This function works with all C++ Standard Library containers and with initializer_list.

You can use this member function in place of the `begin()` template function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container or `initializer_list` of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Retrieves a const (read-only) iterator to the element that follows the last element in the specified container.

C++

```
template <class Container>
auto cend(const Container& cont)
   -> decltype(cont.end());
```

Parameters

cont

A container or `initializer_list`.

Return Value

A constant `cont.end()`.

Remarks

This function works with all C++ Standard Library containers and with initializer_list.

You can use this member function in place of the end() template function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container or `initializer_list` of any kind that supports `end()` and `cend()`.

```
C++

auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

`crbegin`

Get a reverse read-only iterator to the elements of the container, starting at the end of the container.

```
C++

template <class C> constexpr auto crbegin(const C& c) ->
decltype(std::rbegin(c));
```

Parameters

C

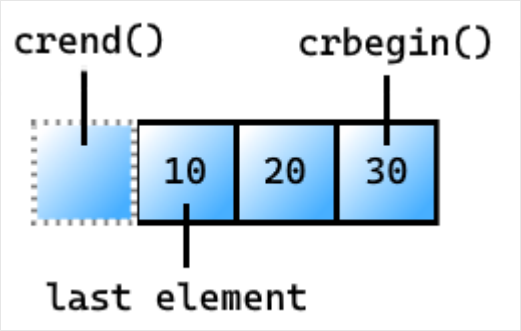
The type of the container.

`c`

A container instance.

Return value

This iterator returns the elements of the container in reverse order, starting at the end of the container.



Example: `crbegin`

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{10, 20, 30};
    for (auto i = std::crbegin(v); i != std::crend(v); ++i)
    {
        std::cout << *i << ' '; // outputs 30 20 10
    }
    // v[1] = 100; // error because the iterator is const
}
```

Output

```
30 20 10
```

`crend`

Get the sentinel at the end of a read-only reversed sequence of elements.

C++

```
template <class C> constexpr auto crend(const C& c) ->
decltype(std::rend(c));
```

Parameters

`C`

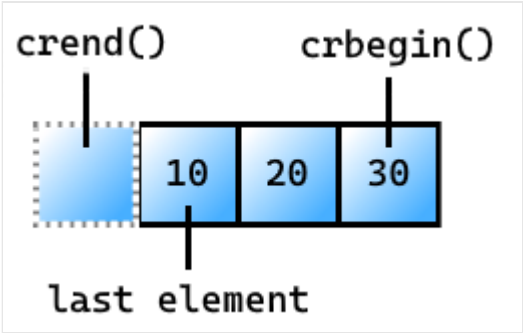
The type of the container.

`c`

A container instance.

Return value

The sentinel follows the last element in a reversed view of the container.



`crend` example

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{10, 20, 30};
    auto vi = std::crend(v);
    --vi; // get off the sentinel and onto the last element in the reversed
range
    std::cout << *vi; // outputs 10
    // vi[0] = 300; // error because the iterator is const
}
```

Output

```
10
```

`data`

Get a pointer to the first element in the container.

```
C++
```

```
1) template <class C> constexpr auto data(C& c) -> decltype(c.data());
2) template <class C> constexpr auto data(const C& c) -> decltype(c.data());
3) template <class T, size_t N> constexpr T* data(T (&array)[N]) noexcept;
4) template <class E> constexpr const E* data(initializer_list<E> il)
noexcept;
```

Parameters

`C`
The type of the container.

`c`
An instance of a container.

`E`
The element type of the initializer list.

`il`
An initializer list.

`N`
The number of elements in the array.

`T`
The type of the data in the array.

Return value

1, 2) A pointer, based on the type of the container, to the first element. For example, if the container is a vector of integers, the type of the return value is an `int *`.

3) A pointer to the first element as an array.

4) A pointer to the first element of the initializer list.

Example `data`

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{ 10, 20, 30 };
    std::string src{ "a string" };

    const char *charPtr = std::data(src);
    int* intPtr = std::data(v);
    std::cout << charPtr << ", " << *intPtr << '\n'; // a string, 10
}
```

Output

```
a string, 10
```

`distance`

Determines the number of increments between the positions addressed by two iterators.

C++

```
template <class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Parameters

`first`

The first iterator whose distance from the second is to be determined.

`last`

The second iterator whose distance from the first is to be determined.

Return Value

The number of times that *`first`* must be incremented until it equals *`last`*.

Remarks

The distance function has constant complexity when `InputIterator` satisfies the requirements for a random-access iterator; otherwise, it has linear complexity and so is potentially expensive.

Example

C++

```
// iterator_distance.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main()
{
    using namespace std;

    list<int> L;
    for (int i = -1; i < 9; ++i)
    {
        L.push_back(2 * i);
    }
    list <int>::iterator L_Iter, LPOS = L.begin();

    cout << "The list L is: ( ";
    for (L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;

    cout << "The iterator LPOS initially points to the first element: "
         << *LPOS << "." << endl;

    advance(LPOS, 7);
    cout << "LPOS is advanced 7 steps forward to point "
        << " to the eighth element: "
        << *LPOS << "." << endl;

    list<int>::difference_type Ldiff;
    Ldiff = distance(L.begin(), LPOS);
    cout << "The distance from L.begin( ) to LPOS is: "
        << Ldiff << "." << endl;
}
```

Output

```
The list L is: ( -2 0 2 4 6 8 10 12 14 16 ).
The iterator LPOS initially points to the first element: -2.
```

```
LPOS is advanced 7 steps forward to point  to the eighth element: 12.
The distance from L.begin( ) to LPOS is: 7.
```

`empty`

C++

```
template <class C> constexpr auto empty(const C& c) -> decltype(c.empty());
template <class T, size_t N> constexpr bool empty(const T (&array)[N])
noexcept;
template <class E> constexpr bool empty(initializer_list<E> il) noexcept;
```

Parameters

C

The type of the container.

c

An instance of a container.

E

The element type of the initializer list.

il

An initializer list.

N

The number of elements in the array.

T

The type of the data in the array.

Return value

Returns `true` if the container has no elements; otherwise `false.`

Example

C++

```
#include <vector>
#include <iostream>
```

```
int main()
{
    std::vector<int> v{ 10,20,30 };
    std::vector<int> v2;

    std::cout << std::boolalpha << std::empty(v); // outputs false
    std::cout << std::boolalpha << ", " << std::empty(v2); // outputs true
}
```

Output

```
false, true
```

end

Retrieves an iterator to the element that follows the last element in the specified container.

C++

```
template <class Container>
auto end(Container& cont)
   -> decltype(cont.end());

template <class Container>
auto end(const Container& cont)
   -> decltype(cont.end());

template <class Ty, class Size>
Ty *end(Ty (& array)[Size]);
```

Parameters

cont

A container.

array

An array of objects of type `Ty`.

Return Value

The first two template functions return `cont.end()` (the first is non-constant and the second is constant).

The third template function returns `array + Size`.

Remarks

For a code example, see begin.

`front_inserter`

Creates an iterator that can insert elements at the front of a specified container.

```
C++
```

```
template <class Container>
front_insert_iterator<Container> front_inserter(Container& Cont);
```

Parameters

`Cont`
The container object whose front is having an element inserted.

Return Value

A `front_insert_iterator` associated with the container object *`Cont`*.

Remarks

The member function front_insert_iterator of the front_insert_iterator class may also be used.

Within the C++ Standard Library, the argument must refer to one of the two sequence containers that have the member function `push_back`: deque Class or "list Class".

Example

```
C++
```

```
// iterator_front_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>
```

```
int main()
{
    using namespace std;

    list<int> L;
    for (int i = -1; i < 9; ++i)
    {
        L.push_back(i);
    }

    cout << "The list L is:\n ( ";
    for (auto L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;

    // Using the template function to insert an element
    front_insert_iterator<list <int>> Iter(L);
    *Iter = 100;

    // Alternatively, you may use the front_insert member function
    front_inserter(L) = 200;

    cout << "After the front insertions, the list L is:\n ( ";
    for (auto L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;
}
```

Output

```
The list L is:
( -1 0 1 2 3 4 5 6 7 8 ).
After the front insertions, the list L is:
( 200 100 -1 0 1 2 3 4 5 6 7 8 ).
```

inserter

A helper template function that lets you use `inserter(Cont, Where)` instead of `insert_iterator<Container>(Cont, Where)`.

C++

```
template <class Container>
insert_iterator<Container>
inserter(
```

```
    Container& Cont,
    typename Container::iterator Where);
```

Parameters

Cont

The container to which new elements are to be added.

Where

An iterator locating the point of insertion.

Remarks

The template function returns insert_iterator`<Container>(Cont, Where)`.

Example

C++

```
// iterator_inserter.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main()
{
    using namespace std;

    list<int> L;
    for (int i = 2; i < 5; ++i)
    {
        L.push_back(10 * i);
    }

    cout << "The list L is:\n ( ";
    for (auto L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;

    // Using the template version to insert an element
    insert_iterator<list<int>> Iter(L, L.begin());
    *Iter = 1;

    // Alternatively, using the member function to insert an element
    inserter(L, L.end()) = 500;
```

```
    cout << "After the insertions, the list L is:\n ( ";
    for (auto L_Iter = L.begin(); L_Iter != L.end(); L_Iter++)
    {
        cout << *L_Iter << " ";
    }
    cout << ")." << endl;
}
```

Output

```
The list L is:
( 20 30 40 ).
After the insertions, the list L is:
( 1 20 30 40 500 ).
```

make_checked_array_iterator

Creates a checked_array_iterator that can be used by other algorithms.

> **Note**
>
> This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

C++

```
template <class Iter>
checked_array_iterator<Iter>
    make_checked_array_iterator(
Iter Ptr,
    size_t Size,
    size_t Index = 0);
```

Parameters

Ptr

A pointer to the destination array.

Size

The size of the destination array.

`Index`

Optional index into the array.

Return Value

An instance of `checked_array_iterator`.

Remarks

The `make_checked_array_iterator` function is defined in the `stdext` namespace.

This function takes a raw pointer—which would ordinarily cause concern about bounds overrun—and wraps it in a checked_array_iterator class that does checking. Because that class is marked as checked, the C++ Standard Library doesn't warn about it. For more information and code examples, see Checked Iterators.

Example

In the following example, a vector is created and populated with 10 items. The contents of the vector are copied into an array by using the copy algorithm, and then `make_checked_array_iterator` is used to specify the destination. This is followed by an intentional violation of the bounds checking so that a debug assertion failure is triggered.

C++

```
// make_checked_array_iterator.cpp
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iterator> // stdext::make_checked_array_iterator
#include <memory> // std::make_unique
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c)
{
    cout << s;

    for (const auto& e : c)
    {
        cout << e << " ";
    }
```

```
    cout << endl;
}

int main()
{
    const size_t dest_size = 10;
    // Old-school but not exception safe, favor make_unique<int[]>
    // int* dest = new int[dest_size];
    unique_ptr<int[]> updest = make_unique<int[]>(dest_size);
    int* dest = updest.get(); // get a raw pointer for the demo

    vector<int> v;

    for (int i = 0; i < dest_size; ++i)
    {
        v.push_back(i);
    }
    print("vector v: ", v);

    copy(v.begin(), v.end(), stdext::make_checked_array_iterator(dest,
dest_size));

    cout << "int array dest: ";
    for (int i = 0; i < dest_size; ++i)
    {
        cout << dest[i] << " ";
    }
    cout << endl;

    // Add another element to the vector to force an overrun.
    v.push_back(10);

    // ! The next line causes a debug assertion when it executes.
    copy(v.begin(), v.end(), stdext::make_checked_array_iterator(dest,
dest_size));
}
```

`make_move_iterator`

Creates a `move iterator` that contains the provided iterator as the `stored` iterator.

C++

```
template <class Iterator>
move_iterator<Iterator>
make_move_iterator(const Iterator& It);
```

Parameters

It

The iterator stored in the new move iterator.

Remarks

The template function returns `move_iterator <Iterator>(_It)`.

`make_unchecked_array_iterator`

Creates an unchecked_array_iterator that can be used by other algorithms.

> **Note**
>
> This function is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

C++

```
template <class Iter>
unchecked_array_iterator<Iter>
    make_unchecked_array_iterator(Iter Ptr);
```

Parameters

Ptr

A pointer to the destination array.

Return Value

An instance of `unchecked_array_iterator`.

Remarks

The `make_unchecked_array_iterator` function is defined in the `stdext` namespace.

This function takes a raw pointer and wraps it in a class that performs no checking and therefore optimizes away to nothing, but it also silences compiler warnings such as C4996. Therefore, this is a targeted way to deal with unchecked-pointer warnings

without globally silencing them or incurring the cost of checking. For more information and code examples, see Checked Iterators.

Example

In the following example, a vector is created and populated with 10 items. The contents of the vector are copied into an array by using the copy algorithm, and then `make_unchecked_array_iterator` is used to specify the destination.

C++

```
// make_unchecked_array_iterator.cpp
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iterator> // stdext::make_unchecked_array_iterator
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c)
{
    cout << s;

    for (const auto& e : c)
    {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    const size_t dest_size = 10;
    int* dest = new int[dest_size];
    vector<int> v;

    for (int i = 0; i < dest_size; ++i)
    {
        v.push_back(i);
    }
    print("vector v: ", v);

    // COMPILER WARNING SILENCED: stdext::unchecked_array_iterator is marked
as checked in debug mode
    // (it performs no checking, so an overrun will trigger undefined
behavior)
    copy(v.begin(), v.end(), stdext::make_unchecked_array_iterator(dest));
```

```
    cout << "int array dest: ";
    for (int i = 0; i < dest_size; ++i)
    {
        cout << dest[i] << " ";
    }
    cout << endl;

    delete[] dest;
}
```

Output

```
vector v: 0 1 2 3 4 5 6 7 8 9
int array dest: 0 1 2 3 4 5 6 7 8 9
```

next

Iterates a specified number of times and returns the new iterator position.

C++

```
template <class InputIterator>
InputIterator next(
    InputIterator first,
    typename iterator_traits<InputIterator>::difference_type off = 1);
```

Parameters

first

The current position.

off

The number of times to iterate.

Return Value

Returns the new iterator position after iterating *off* times.

Remarks

The template function returns `next` incremented *off* times

`prev`

Iterates in reverse a specified number of times and returns the new iterator position.

```
C++
```

```
template <class BidirectionalIterator>
BidirectionalIterator prev(
    BidirectionalIterator first,
    typename iterator_traits<BidirectionalIterator>::difference_type off =
1);
```

Parameters

`first`

The current position.

`off`

The number of times to iterate.

Remarks

The template function returns `next` decremented `off` times.

`rbegin`

Get an iterator, which returns the elements of the container in reverse order.

```
C++
```

```
template <class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template <class C> constexpr auto rbegin(const C& c) ->
decltype(c.rbegin());
```

Parameters

`C`

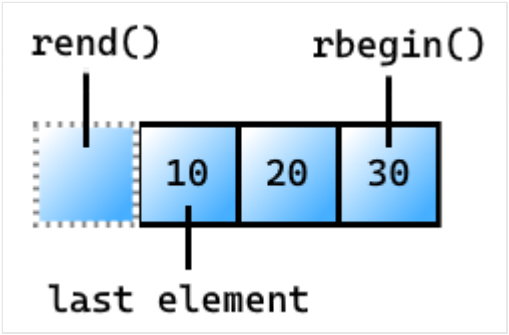
The type of the container.

`c`

A container instance.

Return value

The returned iterator presents the elements of the container in reverse order, starting at the end of the reversed range.



Example `rbegin`

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{ 10, 20, 30 };

    for (auto e = std::rbegin(v); e != std::rend(v); ++e)
    {
        std::cout << *e << ' '; // outputs 30 20 10
    }
}
```

Output

```
30 20 10
```

`rend`

Get the sentinel at the end of a reversed sequence of elements.

C++

```
template <class C> constexpr auto rend(C& c)-> decltype(c.rend());
template <class C> constexpr auto rend(const C& c) -> decltype(c.rend());
```

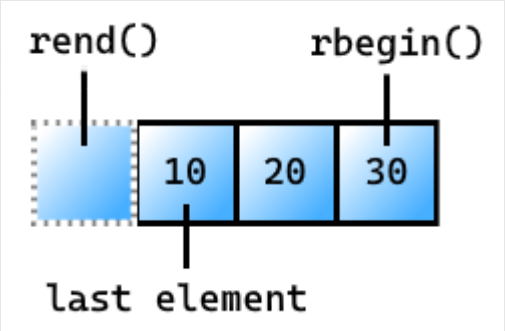
Parameters

`C`
The type of the container.

`c`
A container instance.

Return value

A reverse iterator to the sentinel at the end of the container. The sentinel follows the last element in the reversed view of the container:



`rend` example

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{10, 20, 30};
    auto vi = std::rend(v);
    --vi; // get off the sentinel and onto the last element in the reversed
range
    std::cout << *vi; // outputs 10
}
```

Output

```
10
```

`size`

C++

```
template <class C> constexpr auto size(const C& c)
    -> decltype(c.size());
template <class T, size_t N> constexpr size_t size(const T (&array)[N])
noexcept;
```

Parameters

`C`

The type of the container.

`c`

An instance of a container.

`N`

The number of elements in the array.

`T`

The type of the data in the array.

Return value

The number of elements in the container as an unsigned integer-like value.

Example `size`

C++

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v{ 10, 20, 30 };
    size_t s = std::size(v);
    std::cout << s; // outputs 3
}
```

Output

```
3
```

`<iterator>` operators

Article • 10/03/2022

`operator!=`

Tests if the iterator object on the left side of the operator isn't equal to the iterator object on the right side.

C++

```
template <class RandomIterator>
bool operator!=(const reverse_iterator<RandomIterator>& left, const
reverse_iterator<RandomIterator>& right);

template <class Type, class CharType, class Traits, class Distance>
bool operator!=(const istream_iterator<Type, CharType, Traits, Distance>&
left, const istream_iterator<Type, CharType, Traits, Distance>& right);

template <class CharType, class Tr>
bool operator!=(const istreambuf_iterator<CharType, Traits>& left, const
istreambuf_iterator<CharType, Traits>& right);
```

Parameters

`left`

An object of type `iterator`.

`right`

An object of type `iterator`.

Return Value

`true` if the iterator objects aren't equal; **`false`** if the iterator objects are equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they aren't equal.

Example

C++

```
/// iterator_op_ne.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 1; i < 9; ++i)
    {
        vec.push_back(i);
    }

    cout << "The vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin();

    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 != rVPOS2)
    {
        cout << "The iterators are not equal." << endl;
    }
    else
    {
        cout << "The iterators are equal." << endl;
    }

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 != rVPOS2)
    {
        cout << "The iterators are not equal." << endl;
    }
    else
    {
```

```
        cout << "The iterators are equal." << endl;
    }
}
```

Output

```
The vector vec is: ( 1 2 3 4 5 6 7 8 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 8.
The iterators are equal.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 7.
The iterators are not equal.
```

`operator==`

Tests if the iterator object on the left side of the operator is equal to the iterator object on the right side.

C++

```
template <class RandomIterator1, class RandomIterator2>
bool operator==(
    const move_iterator<RandomIterator1>& left,
    const move_iterator<RandomIterator2>& right);

template <class RandomIterator1, class RandomIterator2>
bool operator==(
    const reverse_iterator<RandomIterator1>& left,
    const reverse_iterator<RandomIterator2>& right);

template <class Type, class CharType, class Traits, class Distance>
bool operator==(
    const istream_iterator<Type, CharType, Traits, Distance>& left,
    const istream_iterator<Type, CharType, Traits, Distance>& right);

template <class CharType, class Tr>
bool operator==(
    const istreambuf_iterator<CharType, Traits>& left,
    const istreambuf_iterator<CharType, Traits>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator objects are equal; **false** if the iterator objects aren't equal.

Remarks

One iterator object is equal to another if they address the same elements in a container. If two iterators point to different elements in a container, then they aren't equal.

The first two template operators return true only if both *left* and *right* store the same iterator. The third template operator returns true only if both *left* and *right* store the same stream pointer. The fourth template operator returns `left.equal (right)`.

Example

C++

```
// iterator_op_eq.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 1; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin();
```

```
    cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 == rVPOS2)
    {
        cout << "The iterators are equal." << endl;
    }
    else
    {
        cout << "The iterators are not equal." << endl;
    }

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 == rVPOS2)
    {
        cout << "The iterators are equal." << endl;
    }
    else
    {
        cout << "The iterators are not equal." << endl;
    }
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
The iterators are equal.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 8.
The iterators are not equal.
```

operator<

Tests if the iterator object on the left side of the operator is less than the iterator object on the right side.

C++

```
template <class RandomIterator>
bool operator<(const reverse_iterator<RandomIterator>& left, const
reverse_iterator<RandomIterator>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is less than the iterator on the right side of the expression; **false** if it's greater than or equal to the iterator on the right.

Remarks

One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object isn't less than another if it addresses either the same element as the other iterator object or an element that occurs later in the container than the element addressed by the other iterator object.

Example

C++

```
// iterator_op_lt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
```

```
    }
    cout << ")." << endl;

    // Initializing reverse_iterators to the last element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin();

    cout << "The iterators rVPOS1& rVPOS2 initially point to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 < rVPOS2)
    {
        cout << "The iterator rVPOS1 is less than"
            << " the iterator rVPOS2." << endl;
    }
    else
    {
        cout << "The iterator rVPOS1 is not less than"
            << " the iterator rVPOS2." << endl;

        rVPOS2++;
        cout << "The iterator rVPOS2 now points to the second "
            << "element\n in the reversed sequence: "
            << *rVPOS2 << "." << endl;

        if (rVPOS1 < rVPOS2)
        {
            cout << "The iterator rVPOS1 is less than"
                << " the iterator rVPOS2." << endl;
        }
        else
        {
            cout << "The iterator rVPOS1 is not less than"
                << " the iterator rVPOS2." << endl;
        }
    }
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1& rVPOS2 initially point to the first element
in the reversed sequence: 10.
The iterator rVPOS1 is not less than the iterator rVPOS2.
The iterator rVPOS2 now points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is less than the iterator rVPOS2.
```

operator<=

Tests if the iterator object on the left side of the operator is less than or equal to the iterator object on the right side.

```
C++
```

```
template <class RandomIterator>
bool operator<=(const reverse_iterator<RandomIterator>& left, const
reverse_iterator<RandomIterator>& right);
```

Parameters

left

An object of type iterator.

right

An object of type iterator.

Return Value

true if the iterator on the left side of the expression is less than or equal to the iterator on the right side of the expression; **false** if it's greater than the iterator on the right.

Remarks

One iterator object is less than or equal to another if it addresses the same element or an element that occurs earlier in the container than the element addressed by the other iterator object. One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object.

Example

```
C++
```

```
// iterator_op_le.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
```

```
    for (int i = 0; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin() + 1,
        rVPOS2 = vec.rbegin();

    cout << "The iterator rVPOS1 initially points to the "
        << "second element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    cout << "The iterator rVPOS2 initially points to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if (rVPOS1 <= rVPOS2)
    {
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    }
    else
    {
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
    }

    rVPOS2++;
    cout << "The iterator rVPOS2 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if (rVPOS1 <= rVPOS2)
    {
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    }
    else
    {
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
    }
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the second element
in the reversed sequence: 8.
The iterator rVPOS2 initially points to the first element
in the reversed sequence: 10.
The iterator rVPOS1 is greater than the iterator rVPOS2.
The iterator rVPOS2 now points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is less than or equal to the iterator rVPOS2.
```

`operator>`

Tests if the iterator object on the left side of the operator is greater than the iterator object on the right side.

C++

```
template <class RandomIterator>
bool operator>(const reverse_iterator<RandomIterator>& left, const
reverse_iterator<RandomIterator>& right);
```

Parameters

`left`

An object of type iterator.

`right`

An object of type iterator.

Return Value

`true` if the iterator on the left side of the expression is greater than the iterator on the right side of the expression; **`false`** if it's less than or equal to the iterator on the right.

Remarks

One iterator object is greater than another if it addresses an element that occurs later in the container than the element addressed by the other iterator object. One iterator object isn't greater than another if it addresses either the same element as the other iterator object or an element that occurs earlier in the container than the element addressed by the other iterator object.

Example

C++

```
// iterator_op_gt.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 6; ++i) {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin();

    cout << "The iterators rVPOS1 & rVPOS2 initially point to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 > rVPOS2)
    {
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
    }
    else
    {
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    }

    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 > rVPOS2)
    {
        cout << "The iterator rVPOS1 is greater than "
            << "the iterator rVPOS2." << endl;
```

```
    }
    else
    {
        cout << "The iterator rVPOS1 is less than or "
            << "equal to the iterator rVPOS2." << endl;
    }
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1 & rVPOS2 initially point to the first element
in the reversed sequence: 10.
The iterator rVPOS1 is less than or equal to the iterator rVPOS2.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is greater than the iterator rVPOS2.
```

`operator>=`

Tests if the iterator object on the left side of the operator is greater than or equal to the iterator object on the right side.

C++

```
template <class RandomIterator>
bool operator>=(const reverse_iterator<RandomIterator>& left, const
reverse_iterator<RandomIterator>& right);
```

Parameters

`left`

An object of type iterator.

`right`

An object of type iterator.

Return Value

`true` if the iterator on the left side of the expression is greater than or equal to the iterator on the right side of the expression; **`false`** if it's less than the iterator on the right.

Remarks

One iterator object is greater than or equal to another if it addresses the same element or an element that occurs later in the container than the element addressed by the other iterator object. One iterator object is less than another if it addresses an element that occurs earlier in the container than the element addressed by the other iterator object.

Example

C++

```
// iterator_op_ge.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin() + 1;

    cout << "The iterator rVPOS1 initially points to the "
        << "first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    cout << "The iterator rVPOS2 initially points to the "
        << "second element\n in the reversed sequence: "
        << *rVPOS2 << "." << endl;

    if (rVPOS1 >= rVPOS2)
    {
        cout << "The iterator rVPOS1 is greater than or "
            << "equal to the iterator rVPOS2." << endl;
    }
```

```
    else
    {
        cout << "The iterator rVPOS1 is less than "
            << "the iterator rVPOS2." << endl;
    }
    rVPOS1++;
    cout << "The iterator rVPOS1 now points to the second "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    if (rVPOS1 >= rVPOS2)
    {
        cout << "The iterator rVPOS1 is greater than or "
            << "equal to the iterator rVPOS2." << endl;
    }
    else
    {
        cout << "The iterator rVPOS1 is less than "
            << "the iterator rVPOS2." << endl;
    }
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
The iterator rVPOS2 initially points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is less than the iterator rVPOS2.
The iterator rVPOS1 now points to the second element
in the reversed sequence: 8.
The iterator rVPOS1 is greater than or equal to the iterator rVPOS2.
```

`operator+`

Adds an offset to an iterator and returns a `move_iterator` or a `reverse_iterator` addressing the inserted element at the new offset position.

C++

```
template <class RandomIterator, class Diff>
move_iterator<RandomIterator>
operator+(
    Diff _Off,
    const move_iterator<RandomIterator>& right);

template <class RandomIterator>
reverse_iterator<RandomIterator>
operator+(
```

```
    Diff _Off,
    const reverse_iterator<RandomIterator>& right);
```

Parameters

Off

The number of positions the const move_iterator or const reverse_iterator is to be offset.

right

The iterator to be offset.

Return Value

Returns the sum *right* + *Off*.

Example

C++

```
// iterator_op_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin();

    cout << "The iterator rVPOS1 initially points to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
```

```
    vector<int>::difference_type diff = 4;
    rVPOS1 = diff + rVPOS1;

    cout << "The iterator rVPOS1 now points to the fifth "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
The iterator rVPOS1 now points to the fifth element
in the reversed sequence: 2.
```

`operator-`

Subtracts one iterator from another and returns the difference.

C++

```
template <class RandomIterator1, class RandomIterator2>
Tdiff operator-(
    const move_iterator<RandomIterator1>& left,
    const move_iterator<RandomIterator2>& right);

template <class RandomIterator1, class RandomIterator2>
Tdiff operator-(
    const reverse_iterator<RandomIterator1>& left,
    const reverse_iterator<RandomIterator2>& right);
```

Parameters

`left`

An iterator.

`right`

An iterator.

Return Value

The difference between two iterators `.`

Remarks

The first template operator returns `left.base() - right.base()`.

The second template operator returns `right.current - left.current`.

`Tdiff` is determined by the type of the returned expression. Otherwise, it's `RandomIterator1::difference_type`.

Example

C++

```
// iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> vec;
    for (int i = 0; i < 6; ++i)
    {
        vec.push_back(2 * i);
    }

    cout << "The initial vector vec is: ( ";
    for (vector <int>::iterator vIter = vec.begin(); vIter != vec.end();
vIter++)
    {
        cout << *vIter << " ";
    }
    cout << ")." << endl;

    vector <int>::reverse_iterator rVPOS1 = vec.rbegin(),
        rVPOS2 = vec.rbegin();

    cout << "The iterators rVPOS1 & rVPOS2 initially point to "
        << "the first element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

    for (int i = 1; i < 5; ++i)
    {
        rVPOS2++;
    }

    cout << "The iterator rVPOS2 now points to the fifth "
        << "element\n in the reversed sequence: "
```

```
        << *rVPOS2 << "." << endl;

    vector<int>::difference_type diff = rVPOS2 - rVPOS1;
    cout << "The difference: rVPOS2 - rVPOS1= "
        << diff << "." << endl;
}
```

Output

```
The initial vector vec is: ( 0 2 4 6 8 10 ).
The iterators rVPOS1 & rVPOS2 initially point to the first element
in the reversed sequence: 10.
The iterator rVPOS2 now points to the fifth element
in the reversed sequence: 2.
The difference: rVPOS2 - rVPOS1= 4.
```

back_insert_iterator Class

Article • 06/09/2022

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the back end of a sequence and thus provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The `back_insert_iterator` class is templatized on the type of container.

Syntax

C++

```
template <class Container>
class back_insert_iterator;
```

Parameters

Container
The type of container into the back of which elements are to be inserted by a `back_insert_iterator`.

Remarks

The container must satisfy the requirements for a back insertion sequence where is it possible to insert elements at the end of the sequence in amortized constant time. C++ Standard Library sequence containers defined by the deque Class, list Class and vector Class provide the needed `push_back` member function and satisfy these requirements. These three containers and strings may each be adapted to use with `back_insert_iterator`s. A `back_insert_iterator` must always be initialized with its container.

Constructors

Constructor	Description
back_insert_iterator	Constructs a `back_insert_iterator` that inserts elements after the last element in a container.

Typedefs

Type name	Description
container_type	A type that provides a container for the `back_insert_iterator`.
reference	A type that provides a reference for the `back_insert_iterator`.

Operators

Operator	Description
operator*	Dereferencing operator used to implement the output iterator expression * `i` = `x` for a back insertion.
operator++	Increments the `back_insert_iterator` to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression * `i` = `x` for a back insertion.

Requirements

Header: <iterator>

Namespace: std

back_insert_iterator::back_insert_iterator

Constructs a `back_insert_iterator` that inserts elements after the last element in a container.

```
C++
explicit back_insert_iterator(Container& _Cont);
```

Parameters

_Cont
The container that the `back_insert_iterator` is to insert an element into.

Return Value

A `back_insert_iterator` for the parameter container.

Example

C++

```
// back_insert_iterator_back_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The initial vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   // Insertions with member function
   back_inserter ( vec ) = 40;
   back_inserter ( vec ) = 50;

   // Alternatively, insertions can be done with template function
   back_insert_iterator<vector<int> > backiter ( vec );
*backiter = 600;
   backiter++;
*backiter = 700;

   cout << "After the insertions, the vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The initial vector vec is: ( 1 2 3 ).
After the insertions, the vector vec is: ( 1 2 3 40 50 600 700 ).
```

back_insert_iterator::container_type

A type that provides a container for the `back_insert_iterator`.

```
C++
```

```
typedef Container
container_type;
```

Remarks

The type is a synonym for the template parameter **Container**.

Example

```
C++
```

```
// back_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back (  i );
   }

   vector <int>::iterator vIter;
   cout << "The original vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> >::container_type vec1 = vec;
   back_inserter ( vec1 ) = 40;

   cout << "After the insertion, the vector is: ( ";
   for ( vIter = vec1.begin ( ) ; vIter != vec1.end ( ); vIter++)
      cout << *vIter << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
The original vector vec is: ( 1 2 3 ).
After the insertion, the vector is: ( 1 2 3 40 ).
```

back_insert_iterator::operator*

Dereferencing operator used to implement the output iterator expression * *i* = *x*.

C++

```
back_insert_iterator<Container>& operator*();
```

Return Value

A reference to the element inserted at the back of the container.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If **Iter** is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and doesn't change the total number of elements in the sequence.

Example

C++

```
// back_insert_iterator_back_insert.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
```

```
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
*backiter = 10;
   backiter++;      // Increment to the next element
*backiter = 20;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 ).
After the insertions, the vector vec becomes: ( 1 2 3 10 20 ).
```

back_insert_iterator::operator++

Increments the `back_insert_iterator` to the next location into which a value may be stored.

C++

```
back_insert_iterator<Container>& operator++();
back_insert_iterator<Container> operator++(int);
```

Return Value

A `back_insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

C++

```
// back_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 3 ; ++i )
   {
      vec.push_back ( 10 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
*backiter = 30;
   backiter++;      // Increment to the next element
*backiter = 40;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The vector vec is: ( 10 20 ).
After the insertions, the vector vec becomes: ( 10 20 30 40 ).
```

back_insert_iterator::operator=

Appends or pushes a value onto the back end of a container.

C++

```
back_insert_iterator<Container>& operator=(typename
Container::const_reference val);
back_insert_iterator<Container>& operator=(typename Container::value_type&&
val);
```

Parameters

val
The value to be inserted into the container.

Return Value

A reference to the last element inserted at the back of the container.

Remarks

The first member operator evaluates `Container.push_back( val)`,

then returns `*this`. The second member operator evaluates

`container->push_back((typename Container::value_type&&)val)`,

then returns `*this`.

Example

C++

```
// back_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
```

```
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> > backiter ( vec );
*backiter = 10;
   backiter++;      // Increment to the next element
*backiter = 20;
   backiter++;

   cout << "After the insertions, the vector vec becomes: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

back_insert_iterator::reference

A type that provides a reference for the `back_insert_iterator`.

C++

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

C++

```
// back_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 4 ; ++i )
```

```
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   back_insert_iterator<vector<int> >::reference
        RefLast = *(vec.end ( ) - 1 );
   cout << "The last element in the vector vec is: "
        << RefLast << "." << endl;
}
```

Output

```
The vector vec is: ( 1 2 3 ).
The last element in the vector vec is: 3.
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

bidirectional_iterator_tag Struct

Article • 08/03/2021

A class that provides a return type for `iterator_category` function that represents a bidirectional iterator.

Syntax

```
C++

struct bidirectional_iterator_tag    : public forward_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument, so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits`< `Iterator`>:: **iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**< **Iter**>:: **iterator_category** when `Iter` describes an object that can serve as a bidirectional iterator.

Example

See random_access_iterator_tag for an example of how to use `bidirectional_iterator_tag`.

Requirements

Header: <iterator>

Namespace: std

See also

forward_iterator_tag Struct
Thread Safety in the C++ Standard Library

checked_array_iterator Class

Article • 12/07/2021

The `checked_array_iterator` class allows you to transform an array or pointer into a checked iterator. Use this class as a wrapper (using the make_checked_array_iterator function) for raw pointers or arrays as a targeted way to provide checking and to manage unchecked pointer warnings instead of globally silencing these warnings. If necessary, you can use the unchecked version of this class, unchecked_array_iterator.

> **Note**
>
> This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension. For an example demonstrating how to write code that does not require the use of this class, see the second example below.

Syntax

C++

```
template <class _Iterator>
class checked_array_iterator;
```

Remarks

This class is defined in the stdext namespace.

For more information and example code on the checked iterator feature, see Checked Iterators.

Examples

The following sample shows how to define and use a checked array iterator.

If the destination is not large enough to hold all the elements being copied, such as would be the case if you changed the line:

C++

```
copy(a, a + 5, checked_array_iterator<int*>(b, 5));
```

to

C++

```
copy(a, a + 5, checked_array_iterator<int*>(b, 4));
```

A runtime error will occur.

C++

```
// compile with: /EHsc /W4 /MTd
#include <algorithm>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[]={0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b, 5));

   cout << "(";
   for (int i = 0 ; i < 5 ; i++)
      cout << " " << b[i];
   cout << " )" << endl;

   // constructor example
   checked_array_iterator<int*> checked_out_iter(b, 5);
   copy(a, a + 5, checked_out_iter);

   cout << "(";
   for (int i = 0 ; i < 5 ; i++)
      cout << " " << b[i];
   cout << " )" << endl;
}
/* Output:
( 0 1 2 3 4 )
( 0 1 2 3 4 )
*/
```

To avoid the need for the `checked_array_iterator` class when using C++ Standard Library algorithms, consider using a `vector` instead of a dynamically allocated array. The following example demonstrates how to do this.

C++

```
// compile with: /EHsc /W4 /MTd

#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    std::vector<int> v(10);
    int *arr = new int[10];
    for (int i = 0; i < 10; ++i)
    {
        v[i] = i;
        arr[i] = i;
    }

    // std::copy(v.begin(), v.end(), arr); will result in
    // warning C4996. To avoid this warning while using int *,
    // use the Microsoft extension checked_array_iterator.
    std::copy(v.begin(), v.end(),
              stdext::checked_array_iterator<int *>(arr, 10));

    // Instead of using stdext::checked_array_iterator and int *,
    // consider using std::vector to encapsulate the array. This will
    // result in no warnings, and the code will be portable.
    std::vector<int> arr2(10);    // Similar to int *arr = new int[10];
    std::copy(v.begin(), v.end(), arr2.begin());

    for (int j = 0; j < arr2.size(); ++j)
    {
        cout << " " << arr2[j];
    }
    cout << endl;

    return 0;
}
/* Output:
0 1 2 3 4 5 6 7 8 9
*/
```

Constructors

Constructor	Description
checked_array_iterator	Constructs a default `checked_array_iterator` or a `checked_array_iterator` from an underlying iterator.

Typedefs

Type name	Description
difference_type	A type that provides the difference between two `checked_array_iterator`s referring to elements within the same container.
pointer	A type that provides a pointer to an element addressed by a `checked_array_iterator`.
reference	A type that provides a reference to an element addressed by a `checked_array_iterator`.

Member functions

Member function	Description
base	Recovers the underlying iterator from its `checked_array_iterator`.

Operators

Operator	Description
operator==	Tests two `checked_array_iterator`s for equality.
operator!=	Tests two `checked_array_iterator`s for inequality.
operator<	Tests if the `checked_array_iterator` on the left side of the operator is less than the `checked_array_iterator` on the right side.
operator>	Tests if the `checked_array_iterator` on the left side of the operator is greater than the `checked_array_iterator` on the right side.
operator<=	Tests if the `checked_array_iterator` on the left side of the operator is less than or equal to the `checked_array_iterator` on the right side.
operator>=	Tests if the `checked_array_iterator` on the left side of the operator is greater than or equal to the `checked_array_iterator` on the right side.
operator*	Returns the element that a `checked_array_iterator` addresses.
operator->	Returns a pointer to the element addressed by the `checked_array_iterator`.
operator++	Increments the `checked_array_iterator` to the next element.
operator--	Decrements the `checked_array_iterator` to the previous element.
operator+=	Adds a specified offset to a `checked_array_iterator`.

Operator	Description
operator+	Adds an offset to an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.
operator-=	Decrements a specified offset from a `checked_array_iterator`.
operator-	Decrements an offset from an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.
operator[]	Returns a reference to an element offset from the element addressed by a `checked_array_iterator` by a specified number of positions.

Requirements

Header: <iterator>

Namespace: stdext

checked_array_iterator::base

Recovers the underlying iterator from its `checked_array_iterator`.

```
C++

_Iterator base() const;
```

Remarks

For more information, see Checked Iterators.

Example

```
C++

// checked_array_iterators_base.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main() {
   using namespace std;

   int V1[10];
```

```
   for (int i = 0; i < 10 ; i++)
      V1[i] = i;

   int* bpos;

   stdext::checked_array_iterator<int*> rpos(V1, 10);
   rpos++;

   bpos = rpos.base ( );
   cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
}
/* Output:
The iterator underlying rpos is bpos & it points to: 1.
*/
```

checked_array_iterator::checked_array_iterator

Constructs a default `checked_array_iterator` or a `checked_array _iterator` from an underlying iterator.

C++

```
checked_array_iterator();

checked_array_iterator(
    ITerator ptr,
    size_t size,
    size_t index = 0);
```

Parameters

ptr
A pointer to the array.

size
The size of the array.

index
(Optional) An element in the array, to initialize the iterator. By default, the iterator is initialized to the first element in the array.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_ctor.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << " ";
   cout << endl;

   checked_array_iterator<int*> checked_output_iterator(b,5);
   copy (a, a + 5, checked_output_iterator);
   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << " ";
   cout << endl;

   checked_array_iterator<int*> checked_output_iterator2(b,5,3);
   cout << *checked_output_iterator2 << endl;
}
/* Output:
0 1 2 3 4
0 1 2 3 4
3
*/
```

checked_array_iterator::difference_type

A type that provides the difference between two `checked_array_iterator`s referring to elements within the same container.

C++

```
typedef typename iterator_traits<_Iterator>::difference_type
difference_type;
```

Remarks

The `checked_array_iterator` difference type is the same as the iterator difference type.

See checked_array_iterator::operator[] for a code sample.

For more information, see Checked Iterators.

checked_array_iterator::operator==

Tests two `checked_array_iterator`s for equality.

```
C++
```

```
bool operator==(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right
The `checked_array_iterator` against which to check for equality.

Remarks

For more information, see Checked Iterators.

Example

```
C++
```

```
// checked_array_iterators_opeq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);
```

```
   if (checked_output_iterator2 == checked_output_iterator)
      cout << "checked_array_iterators are equal" << endl;
   else
      cout << "checked_array_iterators are not equal" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 == checked_output_iterator)
      cout << "checked_array_iterators are equal" << endl;
   else
      cout << "checked_array_iterators are not equal" << endl;
}
/* Output:
checked_array_iterators are equal
checked_array_iterators are not equal
*/
```

checked_array_iterator::operator!=

Tests two `checked_array_iterator`s for inequality.

C++

```
bool operator!=(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right
The `checked_array_iterator` against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_opneq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>
```

```
using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 != checked_output_iterator)
      cout << "checked_array_iterators are not equal" << endl;
   else
      cout << "checked_array_iterators are equal" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 != checked_output_iterator)
      cout << "checked_array_iterators are not equal" << endl;
   else
      cout << "checked_array_iterators are equal" << endl;
}
/* Output:
checked_array_iterators are equal
checked_array_iterators are not equal
*/
```

`checked_array_iterator::operator<`

Tests if the `checked_array_iterator` on the left side of the operator is less than the `checked_array_iterator` on the right side.

C++

```
bool operator<(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` against which to check for inequality.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_oplt.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 < checked_output_iterator)
      cout << "checked_output_iterator2 is less than
checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is not less than
checked_output_iterator" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 < checked_output_iterator)
      cout << "checked_output_iterator2 is less than
checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is not less than
checked_output_iterator" << endl;
}
/* Output:
checked_output_iterator2 is not less than checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
*/
```

`checked_array_iterator::operator>`

Tests if the `checked_array_iterator` on the left side of the operator is greater than the `checked_array_iterator` on the right side.

C++

```
bool operator>(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right
The `checked_array_iterator` to compare against.

Remarks

See checked_array_iterator::operator< for a code sample.

For more information, see Checked Iterators.

`checked_array_iterator::operator<=`

Tests if the `checked_array_iterator` on the left side of the operator is less than or equal to the `checked_array_iterator` on the right side.

```
C++
```

```
bool operator<=(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right
The `checked_array_iterator` to compare against.

Remarks

See checked_array_iterator::operator>= for a code sample.

For more information, see Checked Iterators.

`checked_array_iterator::operator>=`

Tests if the `checked_array_iterator` on the left side of the operator is greater than or equal to the `checked_array_iterator` on the right side.

```
C++
```

```
bool operator>=(const checked_array_iterator<_Iterator>& right) const;
```

Parameters

right

The `checked_array_iterator` to compare against.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_opgteq.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*> checked_output_iterator2(b,5);

   if (checked_output_iterator2 >= checked_output_iterator)
      cout << "checked_output_iterator2 is greater than or equal to
checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is less than
checked_output_iterator" << endl;

   copy (a, a + 5, checked_output_iterator);
   checked_output_iterator++;

   if (checked_output_iterator2 >= checked_output_iterator)
      cout << "checked_output_iterator2 is greater than or equal to
checked_output_iterator" << endl;
   else
      cout << "checked_output_iterator2 is less than
checked_output_iterator" << endl;
```

```
}
/* Output:
checked_output_iterator2 is greater than or equal to checked_output_iterator
checked_output_iterator2 is less than checked_output_iterator
*/
```

checked_array_iterator::operator*

Returns the element that a `checked_array_iterator` addresses.

C++

```
reference operator*() const;
```

Return Value

The value of the element addressed by the `checked_array_iterator`.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

using namespace std;
using namespace stdext;

int main() {
   int a[] = {0, 1, 2, 3, 4};
   int b[5];
   pair<int, int> c[1];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   for (int i = 0 ; i < 5 ; i++)
      cout << b[i] << endl;
```

```
    c[0].first = 10;
    c[0].second = 20;

   checked_array_iterator<int*> checked_output_iterator(b,5);
   checked_array_iterator<int*>::pointer p = &(*checked_output_iterator);
   checked_array_iterator<pair<int, int>*> chk_c(c, 1);
   checked_array_iterator<pair<int, int>*>::pointer p_c = &(*chk_c);

   cout << "b[0] = " << *p << endl;
   cout << "c[0].first = " << p_c->first << endl;
}
/* Output:
0
1
2
3
4
b[0] = 0
c[0].first = 10
*/
```

`checked_array_iterator::operator->`

Returns a pointer to the element addressed by the `checked_array_iterator`.

C++

```
pointer operator->() const;
```

Return Value

A pointer to the element addressed by the `checked_array_iterator`.

Remarks

See checked_array_iterator::pointer for a code sample.

For more information, see Checked Iterators.

checked_array_iterator::operator++

Increments the `checked_array_iterator` to the next element.

C++

```
checked_array_iterator& operator++();

checked_array_iterator<_Iterator> operator++(int);
```

Return Value

The first operator returns the preincremented `checked_array_iterator` and the second, the postincrement operator, returns a copy of the incremented `checked_array_iterator`.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_op_plus_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   ++checked_output_iterator;
   cout << *checked_output_iterator << endl;
   checked_output_iterator++;
   cout << *checked_output_iterator << endl;
}
/* Output:
6
3
77
*/
```

checked_array_iterator::operator--

Decrements the `checked_array_iterator` to the previous element.

C++

```
checked_array_iterator<_Iterator>& operator--();

checked_array_iterator<_Iterator> operator--(int);
```

Return Value

The first operator returns the predecremented `checked_array_iterator` and the second, the postdecrement operator, returns a copy of the decremented `checked_array_iterator`.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_op_minus_minus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   checked_output_iterator++;
   cout << *checked_output_iterator << endl;
   checked_output_iterator--;
   cout << *checked_output_iterator << endl;
}
/* Output:
6
3
6
*/
```

checked_array_iterator::operator+=

Adds a specified offset to a `checked_array_iterator`.

```
C++
```

```
checked_array_iterator<_Iterator>& operator+=(difference_type _Off);
```

Parameters

_Off
The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `checked_array_iterator`.

Remarks

For more information, see Checked Iterators.

Example

```
C++
```

```
// checked_array_iterators_op_plus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);

   cout << *checked_output_iterator << endl;
   checked_output_iterator += 3;
   cout << *checked_output_iterator << endl;
}
/* Output:
```

```
6
199
*/
```

checked_array_iterator::operator+

Adds an offset to an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.

C++

```
checked_array_iterator<_Iterator> operator+(difference_type _Off) const;
```

Parameters

_Off
The offset to be added to the `checked_array_iterator`.

Return Value

A `checked_array_iterator` addressing the offset element.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_op_plus.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
   int b[5];
   copy(a, a + 5, checked_array_iterator<int*>(b,5));

   checked_array_iterator<int*> checked_output_iterator(b,5);
```

```
   cout << *checked_output_iterator << endl;
   checked_output_iterator = checked_output_iterator + 3;
   cout << *checked_output_iterator << endl;
}
/* Output:
6
199
*/
```

checked_array_iterator::operator-=

Decrements a specified offset from a `checked_array_iterator`.

C++

```
checked_array_iterator<_Iterator>& operator-=(difference_type _Off);
```

Parameters

_Off
The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `checked_array_iterator`.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_op_minus_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace stdext;
   using namespace std;
   int a[] = {6, 3, 77, 199, 222};
```

```
    int b[5];
    copy(a, a + 5, checked_array_iterator<int*>(b,5));

    checked_array_iterator<int*> checked_output_iterator(b,5);

    checked_output_iterator += 3;
    cout << *checked_output_iterator << endl;
    checked_output_iterator -= 2;
    cout << *checked_output_iterator << endl;
}
/* Output:
199
3
*/
```

checked_array_iterator::operator-

Decrements an offset from an iterator and returns the new `checked_array_iterator` addressing the inserted element at the new offset position.

C++

```
checked_array_iterator<_Iterator> operator-(difference_type _Off) const;

difference_type operator-(const checked_array_iterator& right) const;
```

Parameters

_Off
The offset to be decremented from the `checked_array_iterator`.

Return Value

A `checked_array_iterator` addressing the offset element.

Remarks

For more information, see Checked Iterators.

checked_array_iterator::operator[]

Returns a reference to an element offset from the element addressed by a `checked_array_iterator` by a specified number of positions.

C++

```
reference operator[](difference_type _Off) const;
```

Parameters

_Off
The offset from the `checked_array_iterator` address.

Return Value

The reference to the element offset.

Remarks

For more information, see Checked Iterators.

Example

C++

```
// checked_array_iterators_op_diff.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   int V1[10];

   for (int i = 0; i < 10 ; i++)
      V1[i] = i;

   // Declare a difference type for a parameter
   stdext::checked_array_iterator<int*>::difference_type diff = 2;

   stdext::checked_array_iterator<int*> VChkIter(V1, 10);

   stdext::checked_array_iterator<int*>::reference refrpos = VChkIter
[diff];

   cout << refrpos + 1 << endl;
}
/* Output:
3
*/
```

checked_array_iterator::pointer

A type that provides a pointer to an element addressed by a `checked_array_iterator`.

```
C++
```

```
typedef typename iterator_traits<_Iterator>::pointer pointer;
```

Remarks

See checked_array_iterator::operator* for a code sample.

For more information, see Checked Iterators.

checked_array_iterator::reference

A type that provides a reference to an element addressed by a `checked_array_iterator`.

```
C++
```

```
typedef typename iterator_traits<_Iterator>::reference reference;
```

Remarks

See checked_array_iterator::operator[] for a code sample.

For more information, see Checked Iterators.

See also

<iterator>
C++ Standard Library Reference

forward_iterator_tag Struct

Article • 08/03/2021

A class that provides a return type for **iterator_category** function that represents a forward iterator.

Syntax

```
C++

struct forward_iterator_tag    : public input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find out what is the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits`< `Iterator`> **::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**< **Iter**> **::iterator_category** when **Iter** describes an object that can serve as a forward iterator.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use the **iterator_tag**s.

Requirements

Header: <iterator>

Namespace: std

See also

input_iterator_tag Struct
Thread Safety in the C++ Standard Library

front_insert_iterator Class

Article • 06/17/2022

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into the front of a sequence. Thus it provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence containers. The `front_insert_iterator` class is templatized on the type of container.

Syntax

```
C++

template <class Container>
class front_insert_iterator;
```

Parameters

Container
The type of container into the front of which elements are to be inserted by a `front_insert_iterator`.

Remarks

The container must satisfy the requirements for a front insertion sequence where is it possible to insert elements at the beginning of the sequence in amortized constant time. The C++ Standard Library sequence containers defined by the deque Class and list Class provide the needed `push_front` member function and satisfy these requirements. By contrast, sequence containers defined by the vector Class don't satisfy these requirements and can't be adapted to use with `front_insert_iterator`s. A `front_insert_iterator` must always be initialized with its container.

Constructors

Constructor	Description
front_insert_iterator	Creates an iterator that can insert elements at the front of a specified container object.

Typedefs

Type name	Description
container_type	A type that represents the container into which a front insertion is to be made.
reference	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

Operator	Description
operator*	Dereferencing operator used to implement the output iterator expression * `i` = `x` for a front insertion.
operator++	Increments the `front_insert_iterator` to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression * `i` = `x` for a front insertion.

Requirements

Header: <iterator>

Namespace: std

front_insert_iterator::container_type

A type that represents the container into which a front insertion is to be made.

```
C++

typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter *Container*.

Example

C++

```
// front_insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   front_insert_iterator<list<int> >::container_type L2 = L1;
   front_inserter ( L2 ) = 20;
   front_inserter ( L2 ) = 10;
   front_inserter ( L2 ) = 40;

   list <int>::iterator vIter;
   cout << "The list L2 is: ( ";
   for ( vIter = L2.begin ( ) ; vIter != L2.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;
}
/* Output:
The list L2 is: ( 40 10 20 ).
*/
```

front_insert_iterator::front_insert_iterator

Creates an iterator that can insert elements at the front of a specified container object.

C++

```
explicit front_insert_iterator(Container& _Cont);
```

Parameters

_Cont

The container object into which the `front_insert_iterator` is to insert elements.

Return Value

A `front_insert_iterator` for the parameter container object.

Example

C++

```
// front_insert_iterator_front_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the member function to insert an element
   front_inserter ( L ) = 20;

   // Alternatively, one may use the template function
   front_insert_iterator< list < int> > Iter(L);
*Iter = 30;

   cout << "After the front insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
/* Output:
The list L is:
( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
*/
```

front_insert_iterator::operator*

Dereferences the insert iterator returning the element it addresses.

C++

```
front_insert_iterator<Container>& operator*();
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If `Iter` is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and doesn't change the total number of elements in the sequence.

Example

C++

```
// front_insert_iterator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for ( i = -1 ; i < 9 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   front_insert_iterator< list < int> > Iter(L);
*Iter = 20;

   // Alternatively, you may use
   front_inserter ( L ) = 30;

   cout << "After the front insertions, the list L is:\n ( ";
```

```
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
/* Output:
The list L is:
( -2 0 2 4 6 8 10 12 14 16 ).
After the front insertions, the list L is:
( 30 20 -2 0 2 4 6 8 10 12 14 16 ).
*/
```

front_insert_iterator::operator++

Increments the `back_insert_iterator` to the next location into which a value may be stored.

C++

```
front_insert_iterator<Container>& operator++();

front_insert_iterator<Container> operator++(int);
```

Return Value

A `front_insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

C++

```
// front_insert_iterator_op_incre.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
```

```
   front_insert_iterator<list<int> > iter ( L1 );
*iter = 10;
   iter++;
*iter = 20;
   iter++;
*iter = 30;
   iter++;

   list <int>::iterator vIter;
   cout << "The list L1 is: ( ";
   for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
/* Output:
The list L1 is: ( 30 20 10 ).
*/
```

front_insert_iterator::operator=

Appends (pushes) a value onto the front of the container.

C++

```
front_insert_iterator<Container>& operator=(typename
Container::const_reference val);

front_insert_iterator<Container>& operator=(typename Container::value_type&&
val);
```

Parameters

val
The value to be assigned to the container.

Return Value

A reference to the last element inserted at the front of the container.

Remarks

The first member operator evaluates `container.push_front( val)`, then returns `*this`.

The second member operator evaluates

`container->push_front((typename Container::value_type&&) val)`,

then returns `*this`.

Example

C++

```
// front_insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   front_insert_iterator<list<int> > iter ( L1 );
*iter = 10;
   iter++;
*iter = 20;
   iter++;
*iter = 30;
   iter++;

   list <int>::iterator vIter;
   cout << "The list L1 is: ( ";
   for ( vIter = L1.begin ( ) ; vIter != L1.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
/* Output:
The list L1 is: ( 30 20 10 ).
*/
```

front_insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

C++

```
typedef typename Container::reference reference;
```

Example

C++

```
// front_insert_iterator_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L;
   front_insert_iterator<list<int> > fiivIter( L );
*fiivIter = 10;
*fiivIter = 20;
*fiivIter = 30;

   list<int>::iterator LIter;
   cout << "The list L is: ( ";
   for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++)
      cout << *LIter << " ";
   cout << ")." << endl;

   front_insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
   cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
/* Output:
The list L is: ( 30 20 10 ).
The first element in the list L is: 30.
*/
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

input_iterator_tag Struct

Article • 10/20/2023

A class that provides a return type for `iterator_category` function that represents an input iterator.

Syntax

```
C++

struct input_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits`< `Iterator`> **::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**< **Iter**> **::iterator_category** when `Iter` describes an object that can serve as an input iterator.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use `iterator_tag`s.

Requirements

Header: <iterator>

Namespace: std

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

insert_iterator Class

Article • 06/17/2022

Describes an iterator adaptor that satisfies the requirements of an output iterator. It inserts, rather than overwrites, elements into a sequence. Thus it provides semantics that are different from the overwrite semantics provided by the iterators of the C++ sequence and associative containers. The `insert_iterator` class is templatized on the type of container being adapted.

Syntax

```
C++

template <class Container>
class insert_iterator;
```

Parameters

Container
The type of container into which elements are to be inserted by an `insert_iterator`.

Remarks

The container of type `Container` must satisfy the requirements for a variable-sized container and have a two-argument insert member function where the parameters are of type `Container::iterator` and `Container::value_type` and that returns a type `Container::iterator`. C++ Standard Library sequence and sorted associative containers satisfy these requirements and can be adapted to use with `insert_iterator`s. For associative containers, the position argument is treated as a hint, which has the potential to improve or degrade performance depending on how good the hint is. An `insert_iterator` must always be initialized with its container.

Constructors

Constructor	Description
insert_iterator	Constructs an `insert_iterator` that inserts an element into a specified position in a container.

Typedefs

Type name	Description
container_type	A type that represents the container into which a general insertion is to be made.
reference	A type that provides a reference to an element in a sequence controlled by the associated container.

Operators

Operator	Description
operator*	Dereferencing operator used to implement the output iterator expression * `i` = `x` for a general insertion.
operator++	Increments the `insert_iterator` to the next location into which a value may be stored.
operator=	Assignment operator used to implement the output iterator expression * `i` = `x` for a general insertion.

Requirements

Header: <iterator>

Namespace: std

insert_iterator::container_type

A type that represents the container into which a general insertion is to be made.

```
C++
```

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter *Container*.

Example

C++

```
// insert_iterator_container_type.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   list<int> L1;
   insert_iterator<list<int> >::container_type L2 = L1;
   inserter ( L2, L2.end ( ) ) = 20;
   inserter ( L2, L2.end ( ) ) = 10;
   inserter ( L2, L2.begin ( ) ) = 40;

   list <int>::iterator vIter;
   cout << "The list L2 is: ( ";
   for ( vIter = L2.begin ( ) ; vIter != L2.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
/* Output:
The list L2 is: ( 40 20 10 ).
*/
```

insert_iterator::insert_iterator

Constructs an `insert_iterator` that inserts an element into a specified position in a container.

C++

```
insert_iterator(Container& _Cont, typename Container::iterator _It);
```

Parameters

_Cont
The container into which the `insert_iterator` is to insert elements.

_It
The position for the insertion.

Remarks

All containers have the insert member function called by the `insert_iterator`. For associative containers the position parameter is merely a suggestion. The inserter function provides a convenient way to insert to values.

Example

C++

```
// insert_iterator_insert_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 1 ; i < 4 ; ++i )
   {
      L.push_back ( 10 * i );
   }

   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;

   // Using the member function to insert an element
   inserter ( L, L.begin ( ) ) = 2;

   // Alternatively, you may use the template version
   insert_iterator< list < int> > Iter(L, L.end ( ) );
*Iter = 300;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
/* Output:
The list L is:
( 10 20 30 ).
After the insertions, the list L is:
( 2 10 20 30 300 ).
*/
```

insert_iterator::operator*

Dereferences the insert iterator returning the element is addresses.

```
C++
```

```
insert_iterator<Container>& operator*();
```

Return Value

The member function returns the value of the element addressed.

Remarks

Used to implement the output iterator expression ***Iter** = **value**. If `Iter` is an iterator that addresses an element in a sequence, then ***Iter** = **value** replaces that element with value and doesn't change the total number of elements in the sequence.

Example

```
C++
```

```
// insert_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 0 ; i < 4 ; ++i )
   {
      L.push_back ( 2 * i );
   }

   cout << "The original list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   insert_iterator< list < int> > Iter(L, L.begin ( ) );
*Iter = 10;
```

```
*Iter = 20;
*Iter = 30;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
/* Output:
The original list L is:
( 0 2 4 6 ).
After the insertions, the list L is:
( 10 20 30 0 2 4 6 ).
*/
```

insert_iterator::operator++

Increments the `insert_iterator` to the next location into which a value may be stored.

C++

```
insert_iterator<Container>& operator++();

insert_iterator<Container> operator++(int);
```

Parameters

A `insert_iterator` addressing the next location into which a value may be stored.

Remarks

Both preincrementation and postincrementation operators return the same result.

Example

C++

```
// insert_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   int i;

   vector<int> vec;
   for (i = 1 ; i < 5 ; ++i )
   {
      vec.push_back (  i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is:\n ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   insert_iterator<vector<int> > ii ( vec, vec.begin ( ) );
*ii = 30;
   ii++;
*ii = 40;
   ii++;
*ii = 50;

   cout << "After the insertions, the vector vec becomes:\n ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
/* Output:
The vector vec is:
( 1 2 3 4 ).
After the insertions, the vector vec becomes:
( 30 40 50 1 2 3 4 ).
*/
```

insert_iterator::operator=

Inserts a value into the container and returns the iterator updated to point to the new element.

C++

```
insert_iterator<Container>& operator=(
    typename Container::const_reference val,);

insert_iterator<Container>& operator=(
    typename Container::value_type&& val);
```

Parameters

val
The value to be assigned to the container.

Return Value

A reference to the element inserted into the container.

Remarks

The first member operator evaluates

`Iter = container->insert(Iter, val);`

`++Iter;`

then returns `*this`.

The second member operator evaluates

`Iter = container->insert(Iter, std::move(val));`

`++Iter;`

then returns `*this`.

Example

C++

```
// insert_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   list <int>::iterator L_Iter;

   list<int> L;
   for (i = 0 ; i < 4 ; ++i )
   {
      L.push_back ( 2 * i );
   }
```

```
   cout << "The original list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;

   insert_iterator< list < int> > Iter(L, L.begin ( ) );
*Iter = 10;
*Iter = 20;
*Iter = 30;

   cout << "After the insertions, the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++ )
      cout << *L_Iter << " ";
   cout << ")." << endl;
}
/* Output:
The original list L is:
( 0 2 4 6 ).
After the insertions, the list L is:
( 10 20 30 0 2 4 6 ).
*/
```

insert_iterator::reference

A type that provides a reference to an element in a sequence controlled by the associated container.

C++

```
typedef typename Container::reference reference;
```

Remarks

The type describes a reference to an element of the sequence controlled by the associated container.

Example

C++

```
// insert_iterator_container_reference.cpp
// compile with: /EHsc
#include <iterator>
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;

   list<int> L;
   insert_iterator<list<int> > iivIter( L , L.begin ( ) );
*iivIter = 10;
*iivIter = 20;
*iivIter = 30;

   list<int>::iterator LIter;
   cout << "The list L is: ( ";
   for ( LIter = L.begin ( ) ; LIter != L.end ( ); LIter++ )
      cout << *LIter << " ";
   cout << ")." << endl;

   insert_iterator<list<int> >::reference
        RefFirst = *(L.begin ( ));
   cout << "The first element in the list L is: "
        << RefFirst << "." << endl;
}
/* Output:
The list L is: ( 10 20 30 ).
The first element in the list L is: 10.
*/
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

istream_iterator Class

Article • 06/17/2022

Describes an input iterator object. It extracts objects of class `Type` from an input stream, which it accesses through an object it stores, of type `pointer` to `basic_istream<CharType, Traits>`.

Syntax

C++

```
template <class Type, class CharType = char, class Traits =
char_traits<CharType>, class Distance = ptrdiff_t,>
class istream_iterator
: public iterator<
    input_iterator_tag, Type, Distance,
    const Type *,
    const Type&>;
```

Parameters

Type
The type of object to be extracted from the input stream.

CharType
The type that represents the character type for the `istream_iterator`. This argument is optional and the default value is **`char`**.

Traits
The type that represents the character type for the `istream_iterator`. This argument is optional and the default value is `char_traits<CharType>`.

Distance
A signed integral type that represents the difference type for the `istream_iterator`. This argument is optional and the default value is `ptrdiff_t`.

After constructing or incrementing an object of class istream_iterator with a nonnull stored pointer, the object attempts to extract and store an object of type `Type` from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Constructors

Constructor	Description
istream_iterator	Constructs either an end-of-stream iterator as the default `istream_iterator` or a `istream_iterator` initialized to the iterator's stream type from which it reads.

Typedefs

Type name	Description
char_type	A type that provides for the character type of the `istream_iterator`.
istream_type	A type that provides for the stream type of the `istream_iterator`.
traits_type	A type that provides for the character traits type of the `istream_iterator`.

Operators

Operator	Description
operator*	The dereferencing operator returns the stored object of type `Type` addressed by the `istream_iterator`.
operator->	Returns the value of a member, if any.
operator++	Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

Requirements

Header: <iterator>

Namespace: std

istream_iterator::char_type

A type that provides for the character type of the `istream_iterator`.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `Chartype`.

Example

C++

```
// istream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef istream_iterator<int>::char_type CHT1;
   typedef istream_iterator<int>::traits_type CHTR1;

   // Standard iterator interface for reading
   // elements from the input stream:
   cout << "Enter integers separated by spaces & then\n"
        << " any character ( try example: '2 4 f' ): ";

   // istream_iterator for reading int stream
   istream_iterator<int, CHT1, CHTR1> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int, CHT1, CHTR1> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

istream_iterator::istream_iterator

Constructs either an end-of-stream iterator as the default `istream_iterator` or a `istream_iterator` initialized to the iterator's stream type from which it reads.

C++

```
istream_iterator();

istream_iterator(istream_type& _Istr);
```

Parameters

_Istr
The input stream to be read use to initialize the `istream_iterator`.

Remarks

The First constructor initializes the input stream pointer with a null pointer and creates an end-of-stream iterator. The second constructor initializes the input stream pointer with *&_Istr*, then attempts to extract and store an object of type `Type`.

The end-of-stream iterator can be used to test whether an `istream_iterator` has reached the end of a stream.

Example

C++

```
// istream_iterator_istream_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   // Used in conjunction with copy algorithm
   // to put elements into a vector read from cin
   vector<int> vec ( 4 );
   vector <int>::iterator Iter;

   cout << "Enter 4 integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";
   istream_iterator<int> intvecRead ( cin );

   // Default constructor will test equal to end of stream
   // for delimiting source range of vecor
   copy ( intvecRead , istream_iterator<int>( ) , vec.begin ( ) );
   cin.clear ( );
```

```
   cout << "vec = ";
   for ( Iter = vec.begin( ) ; Iter != vec.end( ) ; Iter++ )
      cout << *Iter << " "; cout << endl;
}
```

istream_iterator::istream_type

A type that provides for the stream type of the `istream_iterator`.

C++

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for `basic_istream`< **CharType**, **Traits**>.

Example

See istream_iterator for an example of how to declare and use `istream_type`.

istream_iterator::operator*

The dereferencing operator returns the stored object of type `Type` addressed by the `istream_iterator`.

C++

```
const Type& operator*() const;
```

Return Value

The stored object of type `Type`.

Example

C++

```
// istream_iterator_operator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Enter integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";

   // istream_iterator from stream cin
   istream_iterator<int> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

`istream_iterator::operator->`

Returns the value of a member, if any.

C++

```
const Type* operator->() const;
```

Return Value

The value of a member, if any.

Remarks

`i->m` is equivalent to `(*i).m`

The operator returns `&*this`.

Example

C++

```
// istream_iterator_operator_vm.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>
#include <complex>

using namespace std;

int main( )
{
   cout << "Enter complex numbers separated by spaces & then\n"
        << " a character pair ( try example: '(1,2) (3,4) (a,b)' ): ";

   // istream_iterator from stream cin
   istream_iterator< complex<double> > intRead ( cin );

   // End-of-stream iterator
   istream_iterator<complex<double> > EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading the real part: " << intRead ->real( ) << endl;
      cout << "Reading the imaginary part: " << intRead ->imag( ) << endl;
      ++intRead;
   }
   cout << endl;
}
```

istream_iterator::operator++

Either extracts an incremented object from the input stream or copies the object before incrementing it and returns the copy.

C++

```
istream_iterator<Type, CharType, Traits, Distance>& operator++();

istream_iterator<Type, CharType, Traits, Distance> operator++(int);
```

Return Value

The first member operator returns a reference to the incremented object of type `Type` extracted from the input stream and the second member function returns a copy of the

object.

Example

```
C++
```

```
// istream_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "Enter integers separated by spaces & then\n"
        << " a character ( try example: '2 4 6 8 a' ): ";

   // istream_iterator from stream cin
   istream_iterator<int> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

istream_iterator::traits_type

A type that provides for the character traits type of the `istream_iterator`.

```
C++
```

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter *Traits*.

Example

C++

```
// istream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   typedef istream_iterator<int>::char_type CHT1;
   typedef istream_iterator<int>::traits_type CHTR1;

   // Standard iterator interface for reading
   // elements from the input stream:
   cout << "Enter integers separated by spaces & then\n"
        << " any character ( try example: '10 20 a' ): ";

   // istream_iterator for reading int stream
   istream_iterator<int, CHT1, CHTR1> intRead ( cin );

   // End-of-stream iterator
   istream_iterator<int, CHT1, CHTR1> EOFintRead;

   while ( intRead != EOFintRead )
   {
      cout << "Reading:  " << *intRead << endl;
      ++intRead;
   }
   cout << endl;
}
```

See also

input_iterator_tag Struct

iterator Struct

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

istreambuf_iterator Class

Article • 06/17/2022

The class template istreambuf_iterator describes an input iterator object that extracts character elements from an input stream buffer, which it accesses through an object it stores, of type pointer to `basic_streambuf`< **CharType**, **Traits**>.

Syntax

C++

```
template <class CharType class Traits = char_traits <CharType>>
class istreambuf_iterator
: public iterator<input_iterator_tag, CharType, typename Traits ::off_type,
CharType*, CharType&>
```

Parameters

CharType
The type that represents the character type for the istreambuf_iterator.

Traits
The type that represents the character type for the istreambuf_iterator. This argument is optional and the default value is `char_traits`< *CharType*>.

Remarks

The istreambuf_iterator class must satisfy the requirements for an input iterator.

After constructing or incrementing an object of class istreambuf_iterator with a non-null stored pointer, the object effectively attempts to extract and store an object of type *CharType* from the associated input stream. The extraction may be delayed, however, until the object is dereferenced or copied. If the extraction fails, the object effectively replaces the stored pointer with a null pointer, thus making an end-of-sequence indicator.

Constructors

Constructor	Description

Constructor	Description
istreambuf_iterator	Constructs an `istreambuf_iterator` that is initialized to read characters from the input stream.

Typedefs

Type name	Description
char_type	A type that provides for the character type of the `ostreambuf_iterator`.
int_type	A type that provides an integer type for an `istreambuf_iterator`.
istream_type	A type that provides for the stream type of the `istream_iterator`.
streambuf_type	A type that provides for the stream type of the `istreambuf_iterator`.
traits_type	A type that provides for the character traits type of the `istream_iterator`.

Member functions

Member function	Description
equal	Tests for equality between two input stream buffer iterators.

Operators

Operator	Description
operator*	The dereferencing operator returns the next character in the stream.
operator++	Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.

Requirements

Header: <iterator>

Namespace: std

istreambuf_iterator::char_type

A type that provides for the character type of the `ostreambuf_iterator`.

C++

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

Example

C++

```
// istreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
   using namespace std;

   typedef istreambuf_iterator<char>::char_type CHT1;
   typedef istreambuf_iterator<char>::traits_type CHTR1;

   cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   // istreambuf_iterator for input stream
   istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with dot-separators
   replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}
```

istreambuf_iterator::equal

Tests for equivalence between two input stream buffer iterators.

C++

```
bool equal(const istreambuf_iterator<CharType, Traits>& right) const;
```

Parameters

right
The iterator for which to check for equality.

Return Value

`true` if both `istreambuf_iterator`s are end-of-stream iterators or if neither is an end-of-stream iterator; otherwise `false`.

Remarks

A range is defined by the `istreambuf_iterator` to the current position and the end-of-stream iterator, but since all non-end-of stream iterators are equivalent under the `equal` member function, it isn't possible to define any subranges using `istreambuf_iterator`s. The `==` and `!=` operators have the same semantics.

Example

C++

```
// istreambuf_iterator_equal.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;

   cout << "(Try the example: 'Hello world!'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   istreambuf_iterator<char> charReadIn1 ( cin );
   istreambuf_iterator<char> charReadIn2 ( cin );

   bool b1 = charReadIn1.equal ( charReadIn2 );

   if (b1)
      cout << "The iterators are equal." << endl;
   else
```

```
        cout << "The iterators are not equal." << endl;
}
```

istreambuf_iterator::int_type

A type that provides an integer type for an `istreambuf_iterator`.

C++

```
typedef typename traits_type::int_type int_type;
```

Remarks

The type is a synonym for `Traits::int_type`.

Example

C++

```
// istreambuf_iterator_int_type.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;
   istreambuf_iterator<char>::int_type inttype1 = 100;
   cout << "The inttype1 = " << inttype1 << "." << endl;
}
/* Output:
The inttype1 = 100.
*/
```

istreambuf_iterator::istream_type

A type that provides for the stream type of the `istreambuf_iterator`.

C++

```
typedef basic_istream<CharType, Traits> istream_type;
```

Remarks

The type is a synonym for `basic_istream`< **CharType**, **Traits**>.

Example

See istreambuf_iterator for an example of how to declare and use `istream_type`.

istreambuf_iterator::istreambuf_iterator

Constructs an istreambuf_iterator that is initialized to read characters from the input stream.

```
C++
```

```
istreambuf_iterator(streambuf_type* strbuf = 0) throw();
istreambuf_iterator(istream_type& _Istr) throw();
```

Parameters

strbuf
The input stream buffer to which the `istreambuf_iterator` is being attached.

_Istr
The input stream to which the `istreambuf_iterator` is being attached.

Remarks

The first constructor initializes the input stream-buffer pointer with *strbuf*. The second constructor initializes the input stream-buffer pointer with *_Istr*. `rdbuf`, and then eventually attempts to extract and store an object of type `CharType`.

Example

```
C++
```

```
// istreambuf_iterator_istreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <algorithm>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   // Following declarations will not compile:
   istreambuf_iterator<char>::istream_type &istrm = cin;
   istreambuf_iterator<char>::streambuf_type *strmbf = cin.rdbuf( );

   cout << "(Try the example: 'Oh what a world!'\n"
      << " then an Enter key to insert into the output,\n"
      << " & use a ctrl-Z Enter key combination to exit): ";
   istreambuf_iterator<char> charReadIn ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with hyphen-separators
   replace_copy ( charReadIn , istreambuf_iterator<char>( ),
      charOut , ' ' , '-' );
}
```

istreambuf_iterator::operator*

The dereferencing operator returns the next character in the stream.

C++

```
CharType operator*() const;
```

Return Value

The next character in the stream.

Example

C++

```
// istreambuf_iterator_operator_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   cout << "Type string of characters & enter to output it,\n"
      << " with stream buffer iterators,(try: 'I'll be back.')\n"
      << " repeat as many times as desired,\n"
      << " then keystroke ctrl-Z Enter to exit program: ";
   istreambuf_iterator<char> inpos ( cin );
   istreambuf_iterator<char> endpos;
   ostreambuf_iterator<char> outpos ( cout );
   while ( inpos != endpos )
   {
*outpos = *inpos;   //Put value of outpos equal to inpos
      ++inpos;
      ++outpos;
   }
}
```

istreambuf_iterator::operator++

Either returns the next character from the input stream or copies the object before incrementing it and returns the copy.

C++

```
istreambuf_iterator<CharType, Traits>& operator++();
istreambuf_iterator<CharType, Traits> operator++(int);
```

Return Value

An `istreambuf_iterator` or a reference to an `istreambuf_iterator`.

Remarks

The first operator eventually attempts to extract and store an object of type `CharType` from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

Example

C++

```
// istreambuf_iterator_operator_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <iostream>

int main( )
```

```
{
   using namespace std;

   cout << "Type string of characters & enter to output it,\n"
      << " with stream buffer iterators,(try: 'I'll be back.')\n"
      << " repeat as many times as desired,\n"
      << " then keystroke ctrl-Z Enter to exit program: ";
   istreambuf_iterator<char> inpos ( cin );
   istreambuf_iterator<char> endpos;
   ostreambuf_iterator<char> outpos ( cout );
   while ( inpos != endpos )
   {
*outpos = *inpos;
      ++inpos;   //Increment istreambuf_iterator
      ++outpos;
   }
}
```

istreambuf_iterator::streambuf_type

A type that provides for the stream type of the istreambuf_iterator.

```
C++

typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for `basic_streambuf`< **CharType**, **Traits**>.

Example

See istreambuf_iterator for an example of how to declare and use `istreambuf_type`.

istreambuf_iterator::traits_type

A type that provides for the character traits type of the `istream_iterator`.

```
C++

typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter *Traits*.

Example

C++

```
// istreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <algorithm>

int main( )
{
   using namespace std;

   typedef istreambuf_iterator<char>::char_type CHT1;
   typedef istreambuf_iterator<char>::traits_type CHTR1;

   cout << "(Try the example: 'So many dots to be done'\n"
        << " then an Enter key to insert into the output,\n"
        << " & use a ctrl-Z Enter key combination to exit): ";

   // istreambuf_iterator for input stream
   istreambuf_iterator< CHT1, CHTR1> charInBuf ( cin );
   ostreambuf_iterator<char> charOut ( cout );

   // Used in conjunction with replace_copy algorithm
   // to insert into output stream and replace spaces
   // with dot-separators
   replace_copy ( charInBuf , istreambuf_iterator<char>( ),
        charOut , ' ' , '.' );
}
```

See also

iterator Struct

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

iterator Struct

Article • 08/03/2021

An empty base struct used to ensure that a user-defined iterator class works properly with `iterator_trait`s.

Syntax

```
C++
```

```
struct iterator {
   typedef Category iterator_category;
   typedef Type value_type;
   typedef Distance difference_type;
   typedef Distance distance_type;
   typedef Pointer pointer;
   typedef Reference reference;
   };
```

Remarks

The template struct serves as a base type for all iterators. It defines the member types

- `iterator_category` (a synonym for the template parameter `Category`).
- `value_type` (a synonym for the template parameter `Type`).
- `difference_type` (a synonym for the template parameter `Distance`).
- `distance_type` (a synonym for the template parameter `Distance`)
- `pointer` (a synonym for the template parameter `Pointer`).
- `reference` (a synonym for the template parameter `Reference`).

Note that `value_type` should not be a constant type even if `pointer` points at an object of `const` `Type` and reference designates an object of `const` `Type`.

Example

See iterator_traits for an example of how to declare and use the types in the iterator base class.

Requirements

Header: <iterator>

Namespace: std

See also

<iterator>
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

iterator_traits Struct

Article • 08/03/2021

A template helper struct used to specify all the critical type definitions that an iterator should have.

Syntax

```
C++
```

```
struct iterator_traits {
   typedef typename Iterator::iterator_category iterator_category;
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::difference_type difference_type;
   typedef difference_type distance_type;
   typedef typename Iterator::pointer pointer;
   typedef typename Iterator::reference reference;
   };
```

Remarks

The template struct defines the member types

- `iterator_category`: a synonym for `Iterator::iterator_category`.
- `value_type`: a synonym for `Iterator::value_type`.
- `difference_type`: a synonym for `Iterator::difference_type`.
- `distance_type`: a synonym for `Iterator::difference_type.`
- `pointer`: a synonym for `Iterator::pointer`.
- `reference`: a synonym for `Iterator::reference`.

The partial specializations determine the critical types associated with an object pointer of type **Type** * or **const Type** *.

In this implementation you can also use several template functions that do not make use of partial specialization:

```
C++
```

```
template <class Category, class Type, class Diff>
C _Iter_cat(const iterator<Category, Ty, Diff>&);

template <class Ty>
random_access_iterator_tag _Iter_cat(const Ty *);

template <class Category, class Ty, class Diff>
Ty *val_type(const iterator<Category, Ty, Diff>&);

template <class Ty>
Ty *val_type(const Ty *);

template <class Category, class Ty, class Diff>
Diff *_Dist_type(const iterator<Category, Ty, Diff>&);

template <class Ty>
ptrdiff_t *_Dist_type(const Ty *);
```

which determine several of the same types more indirectly. You use these functions as arguments on a function call. Their sole purpose is to supply a useful class template parameter to the called function.

Example

C++

```
// iterator_traits.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <vector>
#include <list>

using namespace std;

template< class it >
void
function( it i1, it i2 )
{
   iterator_traits<it>::iterator_category cat;
   cout << typeid( cat ).name( ) << endl;
   while ( i1 != i2 )
   {
      iterator_traits<it>::value_type x;
      x = *i1;
      cout << x << " ";
      i1++;
   };
   cout << endl;
};
```

```
int main( )
{
   vector<char> vc( 10,'a' );
   list<int> li( 10 );
   function( vc.begin( ), vc.end( ) );
   function( li.begin( ), li.end( ) );
}
/* Output:
struct std::random_access_iterator_tag
a a a a a a a a a a
struct std::bidirectional_iterator_tag
0 0 0 0 0 0 0 0 0 0
*/
```

Requirements

Header: <iterator>

Namespace: std

See also

<iterator>
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

move_iterator Class

Article • 06/20/2022

Class template `move_iterator` is a wrapper for an iterator. The move_iterator provides the same behavior as the iterator it wraps (stores), except it turns the stored iterator's dereference operator into an rvalue reference, turning a copy into a move. For more information about rvalues, see Rvalue Reference Declarator: &&.

Syntax

```
C++
```

```
class move_iterator;
```

Remarks

The class template describes an object that behaves like an iterator except when dereferenced. It stores a random-access iterator of type `Iterator`, accessed by way of the member function `base()`. All operations on a `move_iterator` are performed directly on the stored iterator, except that the result of `operator*` is implicitly cast to `value_type&&` to make an rvalue reference.

A `move_iterator` might be capable of operations that aren't defined by the wrapped iterator. These operations shouldn't be used.

Constructors

Constructor	Description
move_iterator	The constructor for objects of type `move_iterator`.

Typedefs

Type name	Description
iterator_type	A synonym for the template parameter `RandomIterator`.
iterator_category	A synonym for a longer `typename` expression of the same name, `iterator_category` identifies the general abilities of the iterator.

Type name	Description
value_type	A synonym for a longer `typename` expression of the same name, `value_type` describes what type the iterator elements are.
difference_type	A synonym for a longer `typename` expression of the same name, `difference_type` describes the integral type required to express difference values between elements.
pointer	A synonym for template parameter `RandomIterator`.
reference	A synonym for the `rvalue` reference `value_type&&`.

Member functions

Member function	Description
base	The member function returns the stored iterator wrapped by this `move_iterator`.

Operators

Operator	Description
move_iterator::operator*	Returns `(reference)*base().`
move_iterator::operator++	Increments the stored iterator. Exact behavior depends on whether it's a preincrement or a postincrement operation.
move_iterator::operator--	Decrements the stored iterator. Exact behavior depends on whether it's a predecrement or a postdecrement operation.
move_iterator::operator->	Returns `&**this`.
move_iterator::operator-	Returns `move_iterator(*this) -=` by first subtracting the right-hand value from the current position.
move_iterator::operator[]	Returns `(reference)*(*this + off)`. Allows you to specify an offset from the current base to obtain the value at that location.
move_iterator::operator+	Returns `move_iterator(*this) +=` the value. Allows you to add an offset to the base to obtain the value at that location.
move_iterator::operator+=	Adds the right-hand value to the stored iterator, and returns `*this`.

Operator	Description
move_iterator::operator-=	Subtracts the right-hand value from the stored iterator, and returns `*this`.

Requirements

Header: <iterator>

Namespace: std

move_iterator::base

Returns the stored iterator for this `move_iterator`.

```
C++

RandomIterator base() const;
```

Remarks

The member function returns the stored iterator.

move_iterator::difference_type

The type `difference_type` is a `move_iterator` `typedef` based on the iterator trait `difference_type`, and can be used interchangeably with it.

```
C++

typedef typename iterator_traits<RandomIterator>::difference_type
difference_type;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::pointer`.

move_iterator::iterator_category

The type `iterator_category` is a `move_iterator` **`typedef`** based on the iterator trait `iterator_category`, and can be used interchangeably with it.

```
C++
```

```
typedef typename iterator_traits<RandomIterator>::iterator_category
iterator_category;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::iterator_category`.

move_iterator::iterator_type

The type `iterator_type` is based on the template parameter `RandomIterator` for the class template `move_iterator`, and can be used interchangeably in its place.

```
C++
```

```
typedef RandomIterator iterator_type;
```

Remarks

The type is a synonym for the template parameter `RandomIterator`.

move_iterator::move_iterator

Constructs a move iterator. Uses the parameter as the stored iterator.

```
C++
```

```
move_iterator();
explicit move_iterator(RandomIterator right);
template <class Type>
move_iterator(const move_iterator<Type>& right);
```

Parameters

right
The iterator to use as the stored iterator.

Remarks

The first constructor initializes the stored iterator with its default constructor. The remaining constructors initialize the stored iterator with `base.base()`.

move_iterator::operator+=

Adds an offset to the stored iterator, so that the stored iterator points to the element at the new current location. The operator then moves the new current element.

```
C++
```

```
move_iterator& operator+=(difference_type _Off);
```

Parameters

_Off
An offset to add to the current position to determine the new current position.

Return Value

Returns the new current element.

Remarks

The operator adds *_Off* to the stored iterator. Then returns `*this`.

move_iterator::operator-=

Moves across a specified number of previous elements. This operator subtracts an offset from the stored iterator.

```
C++
```

```
move_iterator& operator-=(difference_type _Off);
```

Parameters

Remarks

The operator evaluates `*this += -_Off`. Then returns `*this`.

move_iterator::operator++

Increments the stored iterator that belongs to this `move_iterator`. The current element is accessed by the postincrement operator. The next element is accessed by the preincrement operator.

```
C++
```

```
move_iterator& operator++();
move_iterator operator++(int);
```

Parameters

Remarks

The first (preincrement) operator increments the stored iterator. Then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, evaluates `++*this`. Then returns the copy.

move_iterator::operator+

Returns the iterator position advanced by any number of elements.

```
C++
```

```
move_iterator operator+(difference_type _Off) const;
```

Parameters

Remarks

The operator returns `move_iterator(*this) += _Off`.

move_iterator::operator[]

Allows array index access to elements across the range of the `move iterator`.

```
C++
```

```
reference operator[](difference_type _Off) const;
```

Parameters

Remarks

The operator returns `(reference)*(*this + _Off)`.

move_iterator::operator--

Pre- and postdecrement member operators perform a decrement on the stored iterator.

```
C++
```

```
move_iterator& operator--();
move_iterator operator--();
```

Parameters

Remarks

The first member operator (predecrement) decrements the stored iterator. Then returns `*this`.

The second (postdecrement) operator makes a copy of `*this`, evaluates `--*this`. Then returns the copy.

move_iterator::operator-

Decrements the stored iterator and returns the indicated value.

```
C++
```

```
move_iterator operator-(difference_type _Off) const;
```

Parameters

Remarks

The operator returns `move_iterator(*this) -= _Off`.

move_iterator::operator*

Dereferences the stored iterator and returns the value. This behaves like an `rvalue reference` and performs a move assignment. The operator transfers the current element out of the base iterator. The element that follows becomes the new current element.

```
C++

reference operator*() const;
```

Remarks

The operator returns `(reference)*base()`.

`move_iterator::operator->`

Like a normal `RandomIterator` `operator->`, it provides access to the fields that belong to the current element.

```
C++

pointer operator->() const;
```

Remarks

The operator returns `&**this`.

move_iterator::pointer

The type `pointer` is a **`typedef`** based on the random iterator `RandomIterator` for `move_iterator`, and can be used interchangeably.

```
C++
```

```
typedef RandomIterator  pointer;
```

Remarks

The type is a synonym for `RandomIterator`.

move_iterator::reference

The type `reference` is a **`typedef`** based on `value_type&&` for `move_iterator`, and can be used interchangeably with `value_type&&`.

```
C++
```

```
typedef value_type&& reference;
```

Remarks

The type is a synonym for `value_type&&`, which is an rvalue reference.

move_iterator::value_type

The type `value_type` is a `move_iterator` **`typedef`** based on the iterator trait `value_type`, and can be used interchangeably with it.

```
C++
```

```
typedef typename iterator_traits<RandomIterator>::value_type   value_type;
```

Remarks

The type is a synonym for the iterator trait `typename iterator_traits<RandomIterator>::value_type`.

See also

<iterator>

Lvalues and Rvalues

Move Constructors and Move Assignment Operators (C++)

C++ Standard Library Reference

ostream_iterator Class

Article • 06/20/2022

The class template ostream_iterator describes an output iterator object that writes successive elements onto the output stream with the extraction `operator <<`.

Syntax

```
C++
```

```
template <class Type class CharType = char class Traits = char_traits
<CharType>>
class ostream_iterator
```

Parameters

Type
The type of object to be inserted into the output stream.

CharType
The type that represents the character type for the `ostream_iterator`. This argument is optional and the default value is **`char`**.

Traits
The type that represents the character type for the `ostream_iterator`. This argument is optional and the default value is `char_traits`< *CharType*>.

The ostream_iterator class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an `ostream_iterator`.

Constructors

Constructor	Description
ostream_iterator	Constructs an `ostream_iterator` that is initialized and delimited to write to the output stream.

Typedefs

Type name	Description
char_type	A type that provides for the character type of the `ostream_iterator`.
ostream_type	A type that provides for the stream type of the `ostream_iterator`.
traits_type	A type that provides for the character traits type of the `ostream_iterator`.

Operators

Operator	Description
operator*	Dereferencing operator used to implement the output iterator expression * `i` = `x`.
operator++	A nonfunctional increment operator that returns an `ostream_iterator` to the same object it addressed before the operation was called.
operator=	Assignment operator used to implement the output iterator expression * `i` = `x` for writing to an output stream.

Requirements

Header: <iterator>

Namespace: std

ostream_iterator::char_type

A type that provides for the character type of the iterator.

```
C++
```

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

```
C++
```

```
// ostream_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef ostream_iterator<int>::char_type CHT1;
   typedef ostream_iterator<int>::traits_type CHTR1;

   // ostream_iterator for stream cout
   // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream:
   cout << "The integers written to the output stream\n"
        << "by intOut are:" << endl;
*intOut = 10;
*intOut = 20;
*intOut = 30;
}
/* Output:
The integers written to the output stream
by intOut are:
10
20
30
*/
```

ostream_iterator::operator*

Dereferencing operator used to implement the output iterator expression * *ii* = *x*.

```
C++
```

```
ostream_iterator<Type, CharType, Traits>& operator*();
```

Return Value

A reference to the `ostream_iterator`.

Remarks

The requirements for an output iterator that the `ostream_iterator` must satisfy require only the expression * *ii* = *t* be valid and says nothing about the **`operator`** or the `operator=` on their own. The member operator in this implementation returns **`*this`**.

Example

C++

```
// ostream_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*intOut = 10;
   intOut++;      // No effect on iterator position
*intOut = 20;
*intOut = 30;
}
/* Output:
Elements written to output stream:
10
20
30
*/
```

ostream_iterator::operator++

A nonfunctional increment operator that returns an `ostream_iterator` to the same object it addressed before the operation was called.

C++

```
ostream_iterator<Type, CharType, Traits>& operator++();
ostream_iterator<Type, CharType, Traits> operator++(int);
```

Return Value

A reference to the `ostream_iterator`.

Remarks

These member operators both return `*this`.

Example

C++

```
// ostream_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*intOut = 10;
   intOut++;      // No effect on iterator position
*intOut = 20;
*intOut = 30;
}
/* Output:
Elements written to output stream:
10
20
30
*/
```

ostream_iterator::operator=

Assignment operator used to implement the output_iterator expression * `i` = `x` for writing to an output stream.

C++

```
ostream_iterator<Type, CharType, Traits>& operator=(const Type& val);
```

Parameters

val
The value of the object of type `Type` to be inserted into the output stream.

Return Value

The operator inserts *val* into the output stream associated with the object, followed by the delimiter specified in the ostream_iterator constructor (if any), and then returns a reference to the `ostream_iterator`.

Remarks

The requirements for an output iterator that the `ostream_iterator` must satisfy require only the expression * `ii` = `t` be valid and says nothing about the operator or the operator= on their own. This member operator returns `*this`.

Example

C++

```
// ostream_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   // with new line delimiter
   ostream_iterator<int> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*intOut = 10;
   intOut++;      // No effect on iterator position
*intOut = 20;
*intOut = 30;
}
```

```
/* Output:
Elements written to output stream:
10
20
30
*/
```

ostream_iterator::ostream_iterator

Constructs an `ostream_iterator` that is initialized and delimited to write to the output stream.

C++

```
ostream_iterator(
    ostream_type& _Ostr);

ostream_iterator(
    ostream_type& _Ostr,
    const CharType* _Delimiter);
```

Parameters

_Ostr
The output stream of type ostream_iterator::ostream_type to be iterated over.

_Delimiter
The delimiter that is inserted into the output stream between values.

Remarks

The first constructor initializes the output stream pointer with `&_Ostr`. The delimiter string pointer designates an empty string.

The second constructor initializes the output stream pointer with `&_Ostr` and the delimiter string pointer with *_Delimiter*.

Example

C++

```
// ostream_iterator_ostream_iterator.cpp
// compile with: /EHsc
```

```
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostream_iterator for stream cout
   ostream_iterator<int> intOut ( cout , "\n" );
*intOut = 10;
   intOut++;
*intOut = 20;
   intOut++;

   int i;
   vector<int> vec;
   for ( i = 1 ; i < 7 ; ++i )
   {
      vec.push_back (  i );
   }

   // Write elements to standard output stream
   cout << "Elements output without delimiter: ";
   copy ( vec.begin ( ), vec.end ( ),
          ostream_iterator<int> ( cout ) );
   cout << endl;

   // Write elements with delimiter " : " to output stream
   cout << "Elements output with delimiter: ";
   copy ( vec.begin ( ), vec.end ( ),
          ostream_iterator<int> ( cout, " : " ) );
   cout << endl;
}
/* Output:
10
20
Elements output without delimiter: 123456
Elements output with delimiter: 1 : 2 : 3 : 4 : 5 : 6 :
*/
```

ostream_iterator::ostream_type

A type that provides for the stream type of the iterator.

C++

```
typedef basic_ostream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for basic_ostream< `CharType`, `Traits` >, a stream class of the iostream hierarchy that defines objects that can be used for writing.

Example

See ostream_iterator for an example of how to declare and use `ostream_type`.

ostream_iterator::traits_type

A type that provides for the character traits type of the iterator.

```
C++
```

```
typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter `Traits`.

Example

```
C++
```

```
// ostream_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // The following not OK, but are just the default values:
   typedef ostream_iterator<int>::char_type CHT1;
   typedef ostream_iterator<int>::traits_type CHTR1;

   // ostream_iterator for stream cout
   // with new line delimiter:
    ostream_iterator<int, CHT1, CHTR1> intOut ( cout , "\n" );

   // Standard iterator interface for writing
   // elements to the output stream:
```

```
    cout << "The integers written to output stream\n"
         << "by intOut are:" << endl;
*intOut = 1;
*intOut = 10;
*intOut = 100;
}
/* Output:
The integers written to output stream
by intOut are:
1
10
100
*/
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

ostreambuf_iterator Class

Article • 08/03/2021

The class template ostreambuf_iterator describes an output iterator object that writes successive character elements onto the output stream with the extraction **operator**>>. The `ostreambuf_iterator`s differ from those of the ostream_iterator Class in having characters instead of a generic type at the type of object being inserted into the output stream.

Syntax

```
C++
```

```
template <class CharType = char class Traits = char_traits <CharType>>
```

Parameters

CharType
The type that represents the character type for the ostreambuf_iterator. This argument is optional and the default value is **`char`**.

Traits
The type that represents the character type for the ostreambuf_iterator. This argument is optional and the default value is `char_traits`< *CharType*>.

Remarks

The ostreambuf_iterator class must satisfy the requirements for an output iterator. Algorithms can be written directly to output streams using an `ostreambuf_iterator`. The class provides a low-level stream iterator that allows access to the raw (unformatted) I/O stream in the form of characters and the ability to bypass the buffering and character translations associated with the high-level stream iterators.

Constructors

Constructor	Description
ostreambuf_iterator	Constructs an `ostreambuf_iterator` that is initialized to write characters to the output stream.

Typedefs

Type name	Description
char_type	A type that provides for the character type of the `ostreambuf_iterator`.
ostream_type	A type that provides for the stream type of the `ostream_iterator`.
streambuf_type	A type that provides for the stream type of the `ostreambuf_iterator`.
traits_type	A type that provides for the character traits type of the `ostream_iterator`.

Member functions

Member function	Description
failed	Tests for failure of an insertion into the output stream buffer.

Operators

Operator	Description
operator*	Dereferencing operator used to implement the output iterator expression * `i` = `x`.
operator++	A nonfunctional increment operator that returns an `ostreambuf_iterator` to the same object it addressed before the operation was called.
operator=	The operator inserts a character into the associated stream buffer.

Requirements

Header: <iterator>

Namespace: std

ostreambuf_iterator::char_type

A type that provides for the character type of the `ostreambuf_iterator`.

```
C++
```

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

C++

```
// ostreambuf_iterator_char_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   typedef ostreambuf_iterator<char>::char_type CHT1;
   typedef ostreambuf_iterator<char>::traits_type CHTR1;

   // ostreambuf_iterator for stream cout
   // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output streambuf:
   cout << "The characters written to the output stream\n"
        << " by charOutBuf are: ";
*charOutBuf = 'O';
   charOutBuf++;
*charOutBuf = 'U';
   charOutBuf++;
*charOutBuf = 'T';
   charOutBuf++;
   cout << "." << endl;
}
/* Output:
The characters written to the output stream
by charOutBuf are: OUT.
*/
```

ostreambuf_iterator::failed

Tests for failure of an insertion into the output stream buffer.

C++

```
bool failed() const throw();
```

Return Value

`true` if no insertion into the output stream buffer has failed earlier; otherwise `false`.

Remarks

The member function returns `true` if, in any prior use of member `operator=`, the call to **subf**_-> `sputc` returned **eof**.

Example

C++

```
// ostreambuf_iterator_failed.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   ostreambuf_iterator<char> charOut ( cout );

*charOut = 'a';
   charOut ++;
*charOut  = 'b';
   charOut ++;
*charOut = 'c';
   cout << " are characters output individually." << endl;

   bool b1 = charOut.failed ( );
   if (b1)
       cout << "At least one insertion failed." << endl;
   else
       cout << "No insertions failed." << endl;
}
/* Output:
abc are characters output individually.
No insertions failed.
*/
```

ostreambuf_iterator::operator*

A nonfunctional dereferencing operator used to implement the output iterator expression * *i* = *x*.

```
C++
```

```
ostreambuf_iterator<CharType, Traits>& operator*();
```

Return Value

The ostreambuf iterator object.

Remarks

This operator functions only in the output iterator expression * *i* = *x* to output characters to stream buffer. Applied to an ostreambuf iterator, it returns the iterator; ***iter** returns **iter**,

Example

```
C++
```

```
// ostreambuf_iterator_op_deref.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*charOutBuf = 'O';
   charOutBuf++;   // no effect on iterator position
*charOutBuf = 'U';
*charOutBuf = 'T';
}
/* Output:
Elements written to output stream:
OUT
*/
```

ostreambuf_iterator::operator++

A nonfunctional increment operator that returns an ostream iterator to the same character it addressed before the operation was called.

```
C++
```

```
ostreambuf_iterator<CharType, Traits>& operator++();
ostreambuf_iterator<CharType, Traits>& operator++(int);
```

Return Value

A reference to the character originally addressed or to an implementation-defined object that is convertible to `ostreambuf_iterator`< **CharType**, **Traits**>.

Remarks

The operator is used to implement the output iterator expression * *i* = *x*.

Example

```
C++
```

```
// ostreambuf_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*charOutBuf = 'O';
   charOutBuf++;      // No effect on iterator position
*charOutBuf = 'U';
*charOutBuf = 'T';
}
```

```
/* Output:
Elements written to output stream:
OUT
*/
```

ostreambuf_iterator::operator=

The operator inserts a character into the associated stream buffer.

C++

```
ostreambuf_iterator<CharType, Traits>& operator=(CharType _Char);
```

Parameters

_Char
The character to be inserted into the stream buffer.

Return Value

A reference to the character inserted into the stream buffer.

Remarks

Assignment operator used to implement the output iterator expression * *i* = *x* for writing to an output stream.

Example

C++

```
// ostreambuf_iterator_op_assign.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   // with new line delimiter
   ostreambuf_iterator<char> charOutBuf ( cout );
```

```
   // Standard iterator interface for writing
   // elements to the output stream
   cout << "Elements written to output stream:" << endl;
*charOutBuf = 'O';
   charOutBuf++;      // No effect on iterator position
*charOutBuf = 'U';
*charOutBuf = 'T';
}
/* Output:
Elements written to output stream:
OUT
*/
```

ostreambuf_iterator::ostreambuf_iterator

Constructs an `ostreambuf_iterator` that is initialized to write characters to the output stream.

C++

```
ostreambuf_iterator(streambuf_type* strbuf) throw();
ostreambuf_iterator(ostream_type& Ostr) throw();
```

Parameters

strbuf
The output streambuf object used to initialize the output stream-buffer pointer.

Ostr
The output stream object used to initialize the output stream-buffer pointer.

Remarks

The first constructor initializes the output stream-buffer pointer with *strbuf*.

The second constructor initializes the output stream-buffer pointer with `Ostr`. `rdbuf`. The stored pointer must not be a null pointer.

Example

C++

```
// ostreambuf_iteratorOstreambuf_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   // ostreambuf_iterator for stream cout
   ostreambuf_iterator<char> charOut ( cout );

*charOut = 'O';
   charOut ++;
*charOut  = 'U';
   charOut ++;
*charOut = 'T';
   cout << " are characters output individually." << endl;

   ostreambuf_iterator<char> strOut ( cout );
   string str = "These characters are being written to the output stream.\n
";
   copy ( str.begin ( ), str. end ( ), strOut );
}
/* Output:
OUT are characters output individually.
These characters are being written to the output stream.
*/
```

ostreambuf_iterator::ostream_type

A type that provides for the stream type of the `ostream_iterator`.

C++

```
typedef basicOstream<CharType, Traits> ostream_type;
```

Remarks

The type is a synonym for `basicOstream`< **CharType**, **Traits**>

Example

See ostreambuf_iterator for an example of how to declare and use `ostream_type`.

ostreambuf_iterator::streambuf_type

A type that provides for the stream type of the `ostreambuf_iterator`.

```
C++

typedef basic_streambuf<CharType, Traits> streambuf_type;
```

Remarks

The type is a synonym for `basic_streambuf`< **CharType**, **Traits**>, a stream class for I/O buffers that becomes `streambuf` when specialized to character type **`char`**.

Example

See ostreambuf_iterator for an example of how to declare and use `streambuf_type`.

ostreambuf_iterator::traits_type

A type that provides for the character traits type of the `ostream_iterator`.

```
C++

typedef Traits traits_type;
```

Remarks

The type is a synonym for the template parameter `Traits`.

Example

```
C++

// ostreambuf_iterator_traits_type.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   typedef ostreambuf_iterator<char>::char_type CHT1;
   typedef ostreambuf_iterator<char>::traits_type CHTR1;

   // ostreambuf_iterator for stream cout
   // with new line delimiter:
    ostreambuf_iterator< CHT1, CHTR1> charOutBuf ( cout );

   // Standard iterator interface for writing
   // elements to the output streambuf:
   cout << "The characters written to the output stream\n"
        << " by charOutBuf are: ";
*charOutBuf = 'O';
   charOutBuf++;
*charOutBuf = 'U';
   charOutBuf++;
*charOutBuf = 'T';
   charOutBuf++;
   cout << "." << endl;
}
/* Output:
The characters written to the output stream
by charOutBuf are: OUT.
*/
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

output_iterator_tag Struct

Article • 10/20/2023

A class that provides a return type for `iterator_category` function that represents an output iterator.

Syntax

```
C++

struct output_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits`< `Iterator`> **::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**< **Iter**> **::iterator_category** when `Iter` describes an object that can serve as a output iterator.

This tag is not parameterized on the `value_type` or `difference_type` for the iterator, as with the other iterator tags, because output iterators do not have either a `value_type` or a `difference_type`.

Example

See iterator_traits or random_access_iterator_tag for an example of how to use `iterator_tag`s.

Requirements

Header: <iterator>

Namespace: std

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

random_access_iterator_tag Struct

Article • 08/03/2021

A class that provides a return type for `iterator_category` function that represents a random-access iterator.

Syntax

C++

```
struct random_access_iterator_tag    : public bidirectional_iterator_tag {};
```

Remarks

The category tag classes are used as compile tags for algorithm selection. The template function needs to find the most specific category of its iterator argument so that it can use the most efficient algorithm at compile time. For every iterator of type `Iterator`, `iterator_traits`< `Iterator`> **::iterator_category** must be defined to be the most specific category tag that describes the iterator's behavior.

The type is the same as **iterator**< **Iter**> **::iterator_category** when `Iter` describes an object that can serve as a random-access iterator.

Example

C++

```
// iterator_rait.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>
#include <list>

using namespace std;

int main( )
{
   vector<int> vi;
   vector<char> vc;
   list<char> lc;
   iterator_traits<vector<int>:: iterator>::iterator_category cati;
```

```
   iterator_traits<vector<char>:: iterator>::iterator_category catc;
   iterator_traits<list<char>:: iterator>::iterator_category catlc;

   // These are both random-access iterators
   cout << "The type of iterator for vector<int> is "
        << "identified by the tag:\n "
        << typeid ( cati ).name( ) << endl;
   cout << "The type of iterator for vector<char> is "
        << "identified by the tag:\n "
        << typeid ( catc ).name( ) << endl;
   if ( typeid ( cati ) == typeid( catc ) )
      cout << "The iterators are the same." << endl << endl;
   else
      cout << "The iterators are not the same." << endl << endl;

   // But the list iterator is bidirectinal, not random access
   cout << "The type of iterator for list<char> is "
        << "identified by the tag:\n "
        << typeid (catlc).name( ) << endl;

   // cout << ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) ) << endl;
   if ( typeid ( vi.begin( ) ) == typeid( vc.begin( ) ) )
      cout << "The iterators are the same." << endl;
   else
      cout << "The iterators are not the same." << endl;
   // A random-access iterator is a bidirectional iterator.
   cout << ( void* ) dynamic_cast< iterator_traits<list<char>:: iterator>
          ::iterator_category* > ( &catc ) << endl;
}
```

Sample Output

The following output is for x86.

Output

```
The type of iterator for vector<int> is identified by the tag:
    struct std::random_access_iterator_tag
The type of iterator for vector<char> is identified by the tag:
    struct std::random_access_iterator_tag
The iterators are the same.

The type of iterator for list<char> is identified by the tag:
    struct std::bidirectional_iterator_tag
The iterators are not the same.
0012FF3B
```

Requirements

Header: <iterator>

Namespace: std

See also

bidirectional_iterator_tag Struct
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

reverse_iterator Class

Article • 06/20/2022

The class template is an iterator adaptor that describes a reverse iterator object that behaves like a random-access or bidirectional iterator, only in reverse. It enables the backward traversal of a range.

Syntax

C++

```
template <class RandomIterator>
class reverse_iterator
```

Parameters

RandomIterator The type that represents the iterator to be adapted to operate in reverse.

Remarks

Existing C++ Standard Library containers also define `reverse_iterator` and `const_reverse_iterator` types and have member functions `rbegin` and `rend` that return reverse iterators. These iterators have overwrite semantics. The `reverse_iterator` adaptor supplements this functionality as it offers insert semantics and can also be used with streams.

The `reverse_iterator` that requires a bidirectional iterator must not call any of the member functions `operator+=`, `operator+`, `operator-=`, `operator-`, or `operator[]`, which may only be used with random-access iterators.

The range of an iterator is [*first*, *last*), where the square bracket on the left indicates the inclusion of *first* and the parenthesis on the right indicates the inclusion of elements up to but excluding *last*. The same elements are included in the reversed sequence [**rev** - *first*, **rev** - *last*) so that if *last* is the one-past-the-end element in a sequence, then the first element **rev** - *first* in the reversed sequence points to *(*last* - 1). The identity that relates all reverse iterators to their underlying iterators is:

&*(**reverse_iterator** (*i*)) == &*(*i* - 1).

In practice, this means that in the reversed sequence the reverse_iterator will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Constructors

Constructor	Description
reverse_iterator	Constructs a default `reverse_iterator` or a `reverse_iterator` from an underlying iterator.

Typedefs

Type name	Description
difference_type	A type that provides the difference between two `reverse_iterator`s referring to elements within the same container.
iterator_type	A type that provides the underlying iterator for a `reverse_iterator`.
pointer	A type that provides a pointer to an element addressed by a `reverse_iterator`.
reference	A type that provides a reference to an element addressed by a `reverse_iterator`.

Member functions

Member function	Description
base	Recovers the underlying iterator from its `reverse_iterator`.

Operators

Operator	Description
operator_star	Returns the element that a `reverse_iterator` addresses.
operator+	Adds an offset to an iterator and returns the new `reverse_iterator` addressing the inserted element at the new offset position.
operator++	Increments the `reverse_iterator` to the next element.

Operator	Description
operator+=	Adds a specified offset from a `reverse_iterator`.
operator-	Subtracts an offset from a `reverse_iterator` and returns a `reverse_iterator` addressing the element at the offset position.
operator--	Decrements the `reverse_iterator` to the previous element.
operator-=	Subtracts a specified offset from a `reverse_iterator`.
operator->	Returns a pointer to the element addressed by the `reverse_iterator`.
operator[]	Returns a reference to an element offset from the element addressed by a `reverse_iterator` by a specified number of positions.

Requirements

Header: <iterator>

Namespace: std

reverse_iterator::base

Recovers the underlying iterator from its `reverse_iterator`.

```
C++

RandomIterator base() const;
```

Return Value

The iterator underlying the `reverse_iterator`.

Remarks

The identity that relates all reverse iterators to their underlying iterators is:

&*(`reverse_iterator` (*i*)) == &*(*i* - 1).

In practice, this means that in the reversed sequence the `reverse_iterator` will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the

sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

C++

```
// reverse_iterator_base.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos, bpos;
   pos = find ( vec.begin ( ), vec.end ( ), 6 );
   cout << "The iterator pos points to: " << *pos << "." << endl;

   typedef reverse_iterator<vector<int>::iterator>::iterator_type
it_vec_int_type;

   reverse_iterator<it_vec_int_type> rpos ( pos );
   cout << "The reverse_iterator rpos points to: " << *rpos
        << "." << endl;

   bpos = rpos.base ( );
   cout << "The iterator underlying rpos is bpos & it points to: "
        << *bpos << "." << endl;
}
```

reverse_iterator::difference_type

A type that provides the difference between two `reverse_iterator`s referring to elements within the same container.

```
C++
```

```
typedef typename iterator_traits<RandomIterator>::difference_type
difference_type;
```

Remarks

The `reverse_iterator` difference type is the same as the iterator difference type.

The type is a synonym for the iterator trait typename `iterator_traits`<**RandomIterator**> ::**pointer**.

Example

See reverse_iterator::operator[] for an example of how to declare and use `difference_type`.

reverse_iterator::iterator_type

A type that provides the underlying iterator for a `reverse_iterator`.

```
C++
```

```
typedef RandomIterator iterator_type;
```

Remarks

The type is a synonym for the template parameter `Iterator`.

Example

See reverse_iterator::base for an example of how to declare and use `iterator_type`.

reverse_iterator::operator*

Returns the element that a reverse_iterator addresses.

```
C++
```

```
reference operator*() const;
```

Return Value

The value of the elements addressed by the reverse_iterator.

Remarks

The operator returns *(**current** - 1).

Example

```
C++
```

```
// reverse_iterator_op_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
```

```
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos, bpos;
   pos = find ( vec.begin ( ), vec.end ( ), 6 );

   // Declare a difference type for a parameter
   // declare a reference return type
   reverse_iterator<vector<int>::iterator>::reference refpos = *pos;
   cout << "The iterator pos points to: " << refpos << "." << endl;
}
```

reverse_iterator::operator+

Adds an offset to an iterator and returns the new `reverse_iterator` addressing the inserted element at the new offset position.

C++

```
reverse_iterator<RandomIterator> operator+(difference_type Off) const;
```

Parameters

Off
The offset to be added to the reverse iterator.

Return Value

A `reverse_iterator` addressing the offset element.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

Example

C++

```
// reverse_iterator_op_add.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
```

```
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   vector <int>::reverse_iterator rVPOS2 =rVPOS1 + 2; // offset added
   cout << "After the +2 offset, the iterator rVPOS2 points\n"
        << " to the 3rd element in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
After the +2 offset, the iterator rVPOS2 points
to the 3rd element in the reversed sequence: 6.
```

reverse_iterator::operator++

Increments the reverse_iterator to the previous element.

C++

```
reverse_iterator<RandomIterator>& operator++();
reverse_iterator<RandomIterator> operator++(int);
```

Return Value

The first operator returns the preincremented `reverse_iterator` and the second, the postincrement operator, returns a copy of the incremented `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a bidirectional iterator.

Example

C++

```
// reverse_iterator_op_incr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i - 1 );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;
```

```
    // Initializing reverse_iterators to the last element
    vector <int>::reverse_iterator rVPOS1 = vec.rbegin( );

    cout << "The iterator rVPOS1 initially points to the first "
         << "element\n in the reversed sequence: "
         << *rVPOS1 << "." << endl;

    rVPOS1++;  // postincrement, preincrement: ++rVPSO1

    cout << "After incrementing, the iterator rVPOS1 points\n"
         << " to the second element in the reversed sequence: "
         << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 1 3 5 7 9 ).
The vector vec reversed is: ( 9 7 5 3 1 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 9.
After incrementing, the iterator rVPOS1 points
to the second element in the reversed sequence: 7.
```

reverse_iterator::operator+=

Adds a specified offset from a reverse_iterator.

C++

```
reverse_iterator<RandomIterator>& operator+=(difference_type Off);
```

Parameters

Off
The offset by which to increment the iterator.

Return Value

A reference to the element addressed by the `reverse_iterator`.

Example

C++

```cpp
// reverse_iterator_op_addoff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the last element
   vector <int>::reverse_iterator rVPOS1 = vec.rbegin ( );

   cout << "The iterator rVPOS1 initially points to the first "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   rVPOS1+=2;   // addition of an offset
   cout << "After the +2 offset, the iterator rVPOS1 now points\n"
        << " to the third element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator rVPOS1 initially points to the first element
in the reversed sequence: 10.
After the +2 offset, the iterator rVPOS1 now points
to the third element in the reversed sequence: 6.
```

reverse_iterator::operator-

Subtracts an offset from a `reverse_iterator` and returns a `reverse_iterator` addressing the element at the offset position.

```
C++
```

```
reverse_iterator<RandomIterator> operator-(difference_type Off) const;
```

Parameters

Off
The offset to be subtracted from the reverse_iterator.

Return Value

A `reverse_iterator` addressing the offset element.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

Example

```
C++
```

```
// reverse_iterator_op_sub.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 3 * i );
   }

   vector <int>::iterator vIter;
```

```
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   vector <int>::reverse_iterator rVPOS2 =rVPOS1 - 2; // offset subtracted
   cout << "After the -2 offset, the iterator rVPOS2 points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS2 << "." << endl;
}
```

Output

```
The vector vec is: ( 3 6 9 12 15 ).
The vector vec reversed is: ( 15 12 9 6 3 ).
The iterator rVPOS1 initially points to the last element
in the reversed sequence: 3.
After the -2 offset, the iterator rVPOS2 points
to the 2nd element from the last in the reversed sequence: 9.
```

reverse_iterator::operator--

Decrements the reverse_iterator to the previous element.

C++

```
reverse_iterator<RandomIterator>& operator--();
reverse_iterator<RandomIterator> operator--(int);
```

Return Value

The first operator returns the predecremented `reverse_iterator` and the second, the postdecrement operator, returns a copy of the decremented `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a bidirectional iterator.

Example

C++

```
// reverse_iterator_op_decr.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i - 1 );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;
   rVPOS1--;  // postdecrement, predecrement: --rVPSO1

   cout << "After the decrement, the iterator rVPOS1 points\n"
        << " to the next-to-last element in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 1 3 5 7 9 ).
The vector vec reversed is: ( 9 7 5 3 1 ).
The iterator rVPOS1 initially points to the last element
in the reversed sequence: 1.
After the decrement, the iterator rVPOS1 points
to the next-to-last element in the reversed sequence: 3.
```

reverse_iterator::operator-=

Subtracts a specified offset from a `reverse_iterator`.

C++

```
reverse_iterator<RandomIterator>& operator-=(difference_type Off);
```

Parameters

Off
The offset to be subtracted from the `reverse_iterator`.

Remarks

This member function may only be used if the `reverse_iterator` satisfies the requirements for a random-access iterator.

The operator evaluates **current** + *Off* then returns `*this`.

Example

C++

```
// reverse_iterator_op_suboff.cpp
// compile with: /EHsc
#include <iterator>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
```

```
   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 3 * i );
   }

   vector <int>::iterator vIter;

   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin( ) ; vIter != vec.end( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   // Initializing reverse_iterators to the first element
   vector <int>::reverse_iterator rVPOS1 = vec.rend ( ) - 1;

   cout << "The iterator rVPOS1 initially points to the last "
        << "element\n in the reversed sequence: "
        << *rVPOS1 << "." << endl;

   rVPOS1-=2;      // Subtraction of an offset
   cout << "After the -2 offset, the iterator rVPOS1 now points\n"
        << " to the 2nd element from the last in the reversed sequence: "
        << *rVPOS1 << "." << endl;
}
```

Output

```
The vector vec is: ( 3 6 9 12 15 ).
The vector vec reversed is: ( 15 12 9 6 3 ).
The iterator rVPOS1 initially points to the last element
in the reversed sequence: 3.
After the -2 offset, the iterator rVPOS1 now points
to the 2nd element from the last in the reversed sequence: 9.
```

`reverse_iterator::operator->`

Returns a pointer to the element addressed by the `reverse_iterator`.

C++

```
pointer operator->() const;
```

Return Value

A pointer to the element addressed by the `reverse_iterator`.

Remarks

The operator returns `&**this`.

Example

C++

```
// reverse_iterator_ptrto.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back(pVector::value_type(1,2));
   vec.push_back(pVector::value_type(3,4));
   vec.push_back(pVector::value_type(5,6));

   pVector::iterator pvIter;
   cout << "The vector vec of integer pairs is:\n( ";
   for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
      cout << "( " << pvIter -> first << ", " << pvIter -> second << ") ";
   cout << ")" << endl << endl;

   pVector::reverse_iterator rpvIter;
   cout << "The vector vec reversed is:\n( ";
   for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++ )
      cout << "( " << rpvIter -> first << ", " << rpvIter -> second << ") ";
   cout << ")" << endl << endl;

   pVector::iterator pos = vec.begin ( );
   pos++;
   cout << "The iterator pos points to:\n( " << pos -> first << ", "
   << pos -> second << " )" << endl << endl;

   pVector::reverse_iterator rpos (pos);

   // Use operator -> with return type: why type int and not int*
```

```
    int fint = rpos -> first;
    int sint = rpos -> second;

    cout << "The reverse_iterator rpos points to:\n( " << fint << ", "
    << sint << " )" << endl;
}
```

Output

```
The vector vec of integer pairs is:
( ( 1, 2) ( 3, 4) ( 5, 6) )

The vector vec reversed is:
( ( 5, 6) ( 3, 4) ( 1, 2) )

The iterator pos points to:
( 3, 4 )

The reverse_iterator rpos points to:
( 1, 2 )
```

reverse_iterator::operator[]

Returns a reference to an element offset from the element addressed by a `reverse_iterator` by a specified number of positions.

C++

```
reference operator[](difference_type Off) const;
```

Parameters

Off
The offset from the `reverse_iterator` address.

Return Value

The reference to the element offset.

Remarks

The operator returns *(**`*this`** + `Off`).

Example

C++

```
// reverse_iterator_ret_ref.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for (i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( 2 * i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos;
   pos = find ( vec.begin ( ), vec.end ( ), 8 );
   reverse_iterator<vector<int>::iterator> rpos ( pos );

   // Declare a difference type for a parameter
   reverse_iterator<vector<int>::iterator>::difference_type diff = 2;

   cout << "The iterator pos points to: " << *pos << "." << endl;
   cout << "The iterator rpos points to: " << *rpos << "." << endl;

   // Declare a reference return type & use operator[]
   reverse_iterator<vector<int>::iterator>::reference refrpos = rpos [diff];
   cout << "The iterator rpos now points to: " << refrpos << "." << endl;
}
```

Output

```
The vector vec is: ( 2 4 6 8 10 ).
The vector vec reversed is: ( 10 8 6 4 2 ).
The iterator pos points to: 8.
The iterator rpos points to: 6.
The iterator rpos now points to: 2.
```

reverse_iterator::pointer

A type that provides a pointer to an element addressed by a `reverse_iterator`.

C++

```
typedef typename iterator_traits<RandomIterator>::pointer pointer;
```

Remarks

The type is a synonym for the iterator trait typename `iterator_traits`<*RandomIterator*> ::**pointer**.

Example

C++

```
// reverse_iterator_pointer.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream>

int main( )
{
   using namespace std;

   typedef vector<pair<int,int> > pVector;
   pVector vec;
   vec.push_back( pVector::value_type( 1,2 ) );
   vec.push_back( pVector::value_type( 3,4 ) );
   vec.push_back( pVector::value_type( 5,6 ) );

   pVector::iterator pvIter;
   cout << "The vector vec of integer pairs is:\n" << "( ";
   for ( pvIter = vec.begin ( ) ; pvIter != vec.end ( ); pvIter++)
      cout << "( " << pvIter -> first << ", " << pvIter -> second << ") ";
   cout << ")" << endl;
```

```
   pVector::reverse_iterator rpvIter;
   cout << "\nThe vector vec reversed is:\n" << "( ";
   for ( rpvIter = vec.rbegin( ) ; rpvIter != vec.rend( ); rpvIter++)
      cout << "( " << rpvIter -> first << ", " << rpvIter -> second << ") ";
   cout << ")" << endl;

   pVector::iterator pos = vec.begin ( );
   pos++;
   cout << "\nThe iterator pos points to:\n"
        << "( " << pos -> first << ", "
        << pos -> second << " )" << endl;

   pVector::reverse_iterator rpos (pos);
   cout << "\nThe iterator rpos points to:\n"
        << "( " << rpos -> first << ", "
        << rpos -> second << " )" << endl;
}
```

Output

```
The vector vec of integer pairs is:
( ( 1, 2) ( 3, 4) ( 5, 6) )

The vector vec reversed is:
( ( 5, 6) ( 3, 4) ( 1, 2) )

The iterator pos points to:
( 3, 4 )

The iterator rpos points to:
( 1, 2 )
```

reverse_iterator::reference

A type that provides a reference to an element addressed by a reverse_iterator.

C++

```
typedef typename iterator_traits<RandomIterator>::reference reference;
```

Remarks

The type is a synonym for the iterator trait typename `iterator_traits<RandomIterator>::reference`.

Example

See reverse_iterator::operator[] or reverse_iterator::operator* for examples of how to declare and use `reference`.

reverse_iterator::reverse_iterator

Constructs a default `reverse_iterator` or a `reverse_iterator` from an underlying iterator.

```
C++
```

```
reverse_iterator();
explicit reverse_iterator(RandomIterator right);

template <class Type>
reverse_iterator(const reverse_iterator<Type>& right);
```

Parameters

right
The iterator that is to be adapted to a `reverse_iterator`.

Return Value

A default `reverse_iterator` or a `reverse_iterator` adapting an underlying iterator.

Remarks

The identity that relates all reverse iterators to their underlying iterators is:

`&*(reverse_iterator (i)) == &*(i-1)`.

In practice, this means that in the reversed sequence the reverse_iterator will refer to the element one position beyond (to the right of) the element that the iterator had referred to in the original sequence. So if an iterator addressed the element 6 in the sequence (2, 4, 6, 8), then the `reverse_iterator` will address the element 4 in the reversed sequence (8, 6, 4, 2).

Example

C++

```
// reverse_iterator_reverse_iterator.cpp
// compile with: /EHsc
#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   vector<int> vec;
   for ( i = 1 ; i < 6 ; ++i )
   {
      vec.push_back ( i );
   }

   vector <int>::iterator vIter;
   cout << "The vector vec is: ( ";
   for ( vIter = vec.begin ( ) ; vIter != vec.end ( ); vIter++)
      cout << *vIter << " ";
   cout << ")." << endl;

   vector <int>::reverse_iterator rvIter;
   cout << "The vector vec reversed is: ( ";
   for ( rvIter = vec.rbegin( ) ; rvIter != vec.rend( ); rvIter++)
      cout << *rvIter << " ";
   cout << ")." << endl;

   vector <int>::iterator pos;
   pos = find ( vec.begin ( ), vec.end ( ), 4 );
   cout << "The iterator pos = " << *pos << "." << endl;

   vector <int>::reverse_iterator rpos ( pos );
   cout << "The reverse_iterator rpos = " << *rpos
        << "." << endl;
}
```

See also

<iterator>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

unchecked_array_iterator Class

Article • 08/03/2021

The `unchecked_array_iterator` class allows you to wrap an array or pointer into an unchecked iterator. Use this class as a wrapper (using the make_unchecked_array_iterator function) for raw pointers or arrays as a targeted way to manage unchecked pointer warnings instead of globally silencing these warnings. If possible, prefer the checked version of this class, checked_array_iterator.

> **Note**
>
> This class is a Microsoft extension of the C++ Standard Library. Code implemented by using this function is not portable to C++ Standard build environments that do not support this Microsoft extension.

Syntax

C++

```
template <class Iterator>
class unchecked_array_iterator;
```

Remarks

This class is defined in the stdext namespace.

This is the unchecked version of the checked_array_iterator Class and supports all the same overloads and members. For more information on the checked iterator feature with code examples, see Checked Iterators.

Requirements

Header: <iterator>

Namespace: stdext

See also

`<limits>`

Article • 12/07/2021

Defines the class template `numeric_limits` and two enumerations concerning floating-point representations and rounding.

Requirements

Header: <limits>

Namespace: std

Remarks

Explicit specializations of the `numeric_limits` class describe many properties of the fundamental types, including the character, integer, and floating-point types and `bool` that are implementation defined rather than fixed by the rules of the C++ language. Properties described in <limits> include accuracy, minimum and maximum sized representations, rounding, and signaling type errors.

Members

Enumerations

Name	Description
float_denorm_style	The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:
float_round_style	The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

Classes

Name	Description
numeric_limits Class	The class template describes arithmetic properties of built-in numerical types.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

`<limits>` enums

Article • 12/07/2021

float_denorm_style

The enumeration describes the various methods that an implementation can choose for representing a denormalized floating-point value — one too small to represent as a normalized value:

```
C++
```

```
enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1    };
```

Return Value

The enumeration returns:

- `denorm_indeterminate` if the presence or absence of denormalized forms cannot be determined at translation time.
- `denorm_absent` if denormalized forms are absent.
- `denorm_present` if denormalized forms are present.

Example

See numeric_limits::has_denorm for an example in which the values of this enumeration may be accessed.

float_round_style

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

```
C++
```

```
enum float_round_style {
    round_indeterminate = -1,
```

```
    round_toward_zero = 0,
    round_to_nearest = 1,
    round_toward_infinity = 2,
    round_toward_neg_infinity = 3    };
```

Return Value

The enumeration returns:

- `round_indeterminate` if the rounding method cannot be determined.
- `round_toward_zero` if the round toward zero.
- `round_to_nearest` if the round to nearest integer.
- `round_toward_infinity` if the round away from zero.
- `round_toward_neg_infinity` if the round to more negative integer.

Example

See numeric_limits::round_style for an example in which the values of this enumeration may be accessed.

numeric_limits Class

Article • 08/03/2021

The class template describes arithmetic properties of built-in numerical types.

Syntax

```
C++
```

```
template <class Type>
    class numeric_limits
```

Parameters

Type
The fundamental element data type whose properties are being tested or queried or set. *Type* can also be declared `const`, `volatile`, or `const volatile`.

Remarks

The header defines explicit specializations for the types `wchar_t`, `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `long long`, `unsigned long long`, `char16_t`, and `char32_t`. For these explicit specializations, the member numeric_limits::is_specialized is `true`, and all relevant members have meaningful values. The program can supply additional explicit specializations. Most member functions of the class describe or test possible implementations of `float`.

For an arbitrary specialization, no members have meaningful values. A member object that does not have a meaningful value stores zero (or `false`) and a member function that does not return a meaningful value returns `Type(0)`.

Static Functions and Constants

Name	Description
denorm_min	Returns the smallest nonzero denormalized value.

Name	Description
digits	Returns the number of radix digits that the type can represent without loss of precision.
digits10	Returns the number of decimal digits that the type can represent without loss of precision.
epsilon	Returns the difference between 1 and the smallest value greater than 1 that the data type can represent.
has_denorm	Tests whether a type allows denormalized values.
has_denorm_loss	Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.
has_infinity	Tests whether a type has a representation for positive infinity.
has_quiet_NaN	Tests whether a type has a representation for a quiet not a number (NAN), which is non-signaling.
has_signaling_NaN	Tests whether a type has a representation for signaling not a number (NAN).
infinity	The representation for positive infinity for a type, if available.
is_bounded	Tests if the set of values that a type may represent is finite.
is_exact	Tests if the calculations done on a type are free of rounding errors.
is_iec559	Tests if a type conforms to IEC 559 standards.
is_integer	Tests if a type has an integer representation.
is_modulo	Tests if a type has a modulo representation.
is_signed	Tests if a type has a signed representation.
is_specialized	Tests if a type has an explicit specialization defined in the class template `numeric_limits`.
lowest	Returns the most negative finite value.
max	Returns the maximum finite value for a type.
max_digits10	Returns the number of decimal digits required to ensure that two distinct values of the type have distinct decimal representations.
max_exponent	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.
max_exponent10	Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

Name	Description
min	Returns the minimum normalized value for a type.
min_exponent	Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.
min_exponent10	Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.
quiet_NaN	Returns the representation of a quiet not a number (NAN) for the type.
radix	Returns the integral base, referred to as radix, used for the representation of a type.
round_error	Returns the maximum rounding error for the type.
round_style	Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.
signaling_NaN	Returns the representation of a signaling not a number (NAN) for the type.
tinyness_before	Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.
traps	Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

denorm_min

Returns the smallest nonzero denormalized value.

```
C++
```

```
static constexpr Type denorm_min() throw();
```

Return Value

The smallest nonzero denormalized value.

Remarks

`long double` is the same as `double` for the C++ compiler.

The function returns the minimum value for the type, which is the same as min if has_denorm is not equal to `denorm_present`.

Example

C++

```
// numeric_limits_denorm_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The smallest nonzero denormalized value" << endl
        << "for float objects is: "
        << numeric_limits<float>::denorm_min( ) << endl;
   cout << "The smallest nonzero denormalized value" << endl
        << "for double objects is: "
        << numeric_limits<double>::denorm_min( ) << endl;
   cout << "The smallest nonzero denormalized value" << endl
        << "for long double objects is: "
        << numeric_limits<long double>::denorm_min( ) << endl;

   // A smaller value will round to zero
   cout << numeric_limits<float>::denorm_min( )/2 <<endl;
   cout << numeric_limits<double>::denorm_min( )/2 <<endl;
   cout << numeric_limits<long double>::denorm_min( )/2 <<endl;
}
```

Output

```
The smallest nonzero denormalized value
for float objects is: 1.4013e-045
The smallest nonzero denormalized value
for double objects is: 4.94066e-324
The smallest nonzero denormalized value
for long double objects is: 4.94066e-324
0
0
0
```

digits

Returns the number of radix digits that the type can represent without loss of precision.

C++

```
static constexpr int digits = 0;
```

Return Value

The number of radix digits that the type can represent without loss of precision.

Remarks

The member stores the number of radix digits that the type can represent without change, which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type.

Example

C++

```
// numeric_limits_digits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::digits <<endl;
   cout << numeric_limits<double>::digits <<endl;
   cout << numeric_limits<long double>::digits <<endl;
   cout << numeric_limits<int>::digits <<endl;
   cout << numeric_limits<__int64>::digits <<endl;
}
```

Output

```
24
53
53
31
63
```

digits10

Returns the number of decimal digits that the type can represent without loss of precision.

C++

```
static constexpr int digits10 = 0;
```

Return Value

The number of decimal digits that the type can represent without loss of precision.

Example

C++

```
// numeric_limits_digits10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::digits10 <<endl;
   cout << numeric_limits<double>::digits10 <<endl;
   cout << numeric_limits<long double>::digits10 <<endl;
   cout << numeric_limits<int>::digits10 <<endl;
   cout << numeric_limits<__int64>::digits10 <<endl;
   float f = (float)99999999;
   cout.precision ( 10 );
   cout << "The float is; " << f << endl;
}
```

Output

```
6
15
15
9
18
The float is; 100000000
```

epsilon

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the data type.

C++

```
static constexpr Type epsilon() throw();
```

Return Value

The difference between 1 and the smallest value greater than 1 that is representable for the data type.

Remarks

The value is FLT_EPSILON for type `float`. `epsilon` for a type is the smallest positive floating-point number *N* such that *N* + `epsilon` + *N* is representable.

Example

C++

```
// numeric_limits_epsilon.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for float objects is: "
        << numeric_limits<float>::epsilon( ) << endl;
   cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for double objects is: "
        << numeric_limits<double>::epsilon( ) << endl;
   cout << "The difference between 1 and the smallest "
        << "value greater than 1" << endl
        << "for long double objects is: "
        << numeric_limits<long double>::epsilon( ) << endl;
}
```

Output

```
The difference between 1 and the smallest value greater than 1
for float objects is: 1.19209e-007
The difference between 1 and the smallest value greater than 1
for double objects is: 2.22045e-016
The difference between 1 and the smallest value greater than 1
for long double objects is: 2.22045e-016
```

has_denorm

Tests whether a type allows denormalized values.

C++

```
static constexpr float_denorm_style has_denorm = denorm_absent;
```

Return Value

An enumeration value of type `const float_denorm_style`, indicating whether the type allows denormalized values.

Remarks

The member stores `denorm_present` for a floating-point type that has denormalized values, effectively a variable number of exponent bits.

Example

C++

```
// numeric_limits_has_denorm.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects allow denormalized values: "
        << numeric_limits<float>::has_denorm
        << endl;
   cout << "Whether double objects allow denormalized values: "
        << numeric_limits<double>::has_denorm
        << endl;
```

```
    cout << "Whether long int objects allow denormalized values: "
         << numeric_limits<long int>::has_denorm
         << endl;
}
```

Output

```
Whether float objects allow denormalized values: 1
Whether double objects allow denormalized values: 1
Whether long int objects allow denormalized values: 0
```

has_denorm_loss

Tests whether loss of accuracy is detected as a denormalization loss rather than as an inexact result.

C++

```
static constexpr bool has_denorm_loss = false;
```

Return Value

`true` if the loss of accuracy is detected as a denormalization loss; `false` if not.

Remarks

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 floating-point representations that can affect some results.

Example

C++

```
// numeric_limits_has_denorm_loss.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;
```

```
int main( )
{
   cout << "Whether float objects can detect denormalized loss: "
        << numeric_limits<float>::has_denorm_loss
        << endl;
   cout << "Whether double objects can detect denormalized loss: "
        << numeric_limits<double>::has_denorm_loss
        << endl;
   cout << "Whether long int objects can detect denormalized loss: "
        << numeric_limits<long int>::has_denorm_loss
        << endl;
}
```

Output

```
Whether float objects can detect denormalized loss: 1
Whether double objects can detect denormalized loss: 1
Whether long int objects can detect denormalized loss: 0
```

has_infinity

Tests whether a type has a representation for positive infinity.

C++

```
static constexpr bool has_infinity = false;
```

Return Value

`true` if the type has a representation for positive infinity; `false` if not.

Remarks

The member returns `true` if is_iec559 is `true`.

Example

C++

```
// numeric_limits_has_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;
```

```
int main( )
{
   cout << "Whether float objects have infinity: "
        << numeric_limits<float>::has_infinity
        << endl;
   cout << "Whether double objects have infinity: "
        << numeric_limits<double>::has_infinity
        << endl;
   cout << "Whether long int objects have infinity: "
        << numeric_limits<long int>::has_infinity
        << endl;
}
```

Output

```
Whether float objects have infinity: 1
Whether double objects have infinity: 1
Whether long int objects have infinity: 0
```

has_quiet_NaN

Tests whether a type has a representation for a quiet not a number (NAN), which is nonsignaling.

C++

```
static constexpr bool has_quiet_NaN = false;
```

Return Value

`true` if the **type** has a representation for a quiet NAN; `false` if not.

Remarks

A quiet NAN is an encoding for not a number, which does not signal its presence in an expression. The return value is `true` if is_iec559 is true.

Example

C++

```
// numeric_limits_has_quiet_nan.cpp
// compile with: /EHsc
```

```cpp
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have quiet_NaN: "
        << numeric_limits<float>::has_quiet_NaN
        << endl;
   cout << "Whether double objects have quiet_NaN: "
        << numeric_limits<double>::has_quiet_NaN
        << endl;
   cout << "Whether long int objects have quiet_NaN: "
        << numeric_limits<long int>::has_quiet_NaN
        << endl;
}
```

Output

```
Whether float objects have quiet_NaN: 1
Whether double objects have quiet_NaN: 1
Whether long int objects have quiet_NaN: 0
```

has_signaling_NaN

Tests whether a type has a representation for signaling not a number (NAN).

C++

```cpp
static constexpr bool has_signaling_NaN = false;
```

Return Value

`true` if the type has a representation for a signaling NAN; `false` if not.

Remarks

A signaling NAN is an encoding for not a number, which signals its presence in an expression. The return value is `true` if is_iec559 is true.

Example

C++

```cpp
// numeric_limits_has_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have a signaling_NaN: "
        << numeric_limits<float>::has_signaling_NaN
        << endl;
   cout << "Whether double objects have a signaling_NaN: "
        << numeric_limits<double>::has_signaling_NaN
        << endl;
   cout << "Whether long int objects have a signaling_NaN: "
        << numeric_limits<long int>::has_signaling_NaN
        << endl;
}
```

Output

```
Whether float objects have a signaling_NaN: 1
Whether double objects have a signaling_NaN: 1
Whether long int objects have a signaling_NaN: 0
```

infinity

The representation of positive infinity for a type, if available.

C++

```cpp
static constexpr Type infinity() throw();
```

Return Value

The representation of positive infinity for a type, if available.

Remarks

The return value is meaningful only if has_infinity is `true`.

Example

C++

```cpp
// numeric_limits_infinity.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << numeric_limits<float>::has_infinity <<endl;
   cout << numeric_limits<double>::has_infinity<<endl;
   cout << numeric_limits<long double>::has_infinity <<endl;
   cout << numeric_limits<int>::has_infinity <<endl;
   cout << numeric_limits<__int64>::has_infinity <<endl;

   cout << "The representation of infinity for type float is: "
        << numeric_limits<float>::infinity( ) <<endl;
   cout << "The representation of infinity for type double is: "
        << numeric_limits<double>::infinity( ) <<endl;
   cout << "The representation of infinity for type long double is: "
        << numeric_limits<long double>::infinity( ) <<endl;
}
```

Output

```
1
1
1
0
0
The representation of infinity for type float is: inf
The representation of infinity for type double is: inf
The representation of infinity for type long double is: inf
```

is_bounded

Tests if the set of values that a type may represent is finite.

C++

```cpp
static constexpr bool is_bounded = false;
```

Return Value

`true` if the type has a bounded set of representable values; `false` if not.

Remarks

All predefined types have a bounded set of representable values and return `true`.

Example

C++

```
// numeric_limits_is_bounded.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have bounded set "
        << "of representable values: "
        << numeric_limits<float>::is_bounded
        << endl;
   cout << "Whether double objects have bounded set "
        << "of representable values: "
        << numeric_limits<double>::is_bounded
        << endl;
   cout << "Whether long int objects have bounded set "
        << "of representable values: "
        << numeric_limits<long int>::is_bounded
        << endl;
   cout << "Whether unsigned char objects have bounded set "
        << "of representable values: "
        << numeric_limits<unsigned char>::is_bounded
        << endl;
}
```

Output

```
Whether float objects have bounded set of representable values: 1
Whether double objects have bounded set of representable values: 1
Whether long int objects have bounded set of representable values: 1
Whether unsigned char objects have bounded set of representable values: 1
```

is_exact

Tests if the calculations done on a type are free of rounding errors.

C++

```
static constexpr bool is_exact = false;
```

Return Value

`true` if the calculations are free of rounding errors; `false` if not.

Remarks

All predefined integer types have exact representations for their values and return `false`. A fixed-point or rational representation is also considered exact, but a floating-point representation is not.

Example

C++

```
// numeric_limits_is_exact.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<float>::is_exact
        << endl;
   cout << "Whether double objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<double>::is_exact
        << endl;
   cout << "Whether long int objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<long int>::is_exact
        << endl;
   cout << "Whether unsigned char objects have calculations "
        << "free of rounding errors: "
        << numeric_limits<unsigned char>::is_exact
        << endl;
}
```

Output

```
Whether float objects have calculations free of rounding errors: 0
Whether double objects have calculations free of rounding errors: 0
Whether long int objects have calculations free of rounding errors: 1
Whether unsigned char objects have calculations free of rounding errors: 1
```

is_iec559

Tests if a type conforms to IEC 559 standards.

C++

```
static constexpr bool is_iec559 = false;
```

Return Value

`true` if the type conforms to the IEC 559 standards; `false` if not.

Remarks

The IEC 559 is an international standard for representing floating-point values and is also known as IEEE 754 in the USA.

Example

C++

```
// numeric_limits_is_iec559.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects conform to iec559 standards: "
        << numeric_limits<float>::is_iec559
        << endl;
   cout << "Whether double objects conform to iec559 standards: "
        << numeric_limits<double>::is_iec559
        << endl;
   cout << "Whether int objects conform to iec559 standards: "
        << numeric_limits<int>::is_iec559
        << endl;
   cout << "Whether unsigned char objects conform to iec559 standards: "
```

```
        << numeric_limits<unsigned char>::is_iec559
        << endl;
}
```

Output

```
Whether float objects conform to iec559 standards: 1
Whether double objects conform to iec559 standards: 1
Whether int objects conform to iec559 standards: 0
Whether unsigned char objects conform to iec559 standards: 0
```

is_integer

Tests if a type has an integer representation.

C++

```
static constexpr bool is_integer = false;
```

Return Value

`true` if the type has an integer representation; `false` if not.

Remarks

All predefined integer types have an integer representation.

Example

C++

```
// numeric_limits_is_integer.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have an integral representation: "
        << numeric_limits<float>::is_integer
        << endl;
   cout << "Whether double objects have an integral representation: "
        << numeric_limits<double>::is_integer
```

```
        << endl;
    cout << "Whether int objects have an integral representation: "
         << numeric_limits<int>::is_integer
         << endl;
    cout << "Whether unsigned char objects have an integral representation: "
         << numeric_limits<unsigned char>::is_integer
         << endl;
}
```

Output

```
Whether float objects have an integral representation: 0
Whether double objects have an integral representation: 0
Whether int objects have an integral representation: 1
Whether unsigned char objects have an integral representation: 1
```

is_modulo

Tests if a **type** has a modulo representation.

C++

```
static constexpr bool is_modulo = false;
```

Return Value

`true` if the type has a modulo representation; `false` if not.

Remarks

A modulo representation is a representation where all results are reduced modulo some value. All predefined unsigned integer types have a modulo representation.

Example

C++

```
// numeric_limits_is_modulo.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;
```

```
int main( )
{
   cout << "Whether float objects have a modulo representation: "
        << numeric_limits<float>::is_modulo
        << endl;
   cout << "Whether double objects have a modulo representation: "
        << numeric_limits<double>::is_modulo
        << endl;
   cout << "Whether signed char objects have a modulo representation: "
        << numeric_limits<signed char>::is_modulo
        << endl;
   cout << "Whether unsigned char objects have a modulo representation: "
        << numeric_limits<unsigned char>::is_modulo
        << endl;
}
```

Output

```
Whether float objects have a modulo representation: 0
Whether double objects have a modulo representation: 0
Whether signed char objects have a modulo representation: 1
Whether unsigned char objects have a modulo representation: 1
```

is_signed

Tests if a type has a signed representation.

C++

```
static constexpr bool is_signed = false;
```

Return Value

`true` if the type has a signed representation; `false` if not.

Remarks

The member stores true for a type that has a signed representation, which is the case for all predefined floating-point and signed integer types.

Example

C++

```cpp
// numeric_limits_is_signaled.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have a signed representation: "
        << numeric_limits<float>::is_signed
        << endl;
   cout << "Whether double objects have a signed representation: "
        << numeric_limits<double>::is_signed
        << endl;
   cout << "Whether signed char objects have a signed representation: "
        << numeric_limits<signed char>::is_signed
        << endl;
   cout << "Whether unsigned char objects have a signed representation: "
        << numeric_limits<unsigned char>::is_signed
        << endl;
}
```

Output

```
Whether float objects have a signed representation: 1
Whether double objects have a signed representation: 1
Whether signed char objects have a signed representation: 1
Whether unsigned char objects have a signed representation: 0
```

is_specialized

Tests if a type has an explicit specialization defined in the class template `numeric_limits`.

C++

```cpp
static constexpr bool is_specialized = false;
```

Return Value

`true` if the type has an explicit specialization defined in the class template; **`false`** if not.

Remarks

All scalar types other than pointers have an explicit specialization defined for class template `numeric_limits`.

Example

C++

```
// numeric_limits_is_specialized.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<float>::is_specialized
        << endl;
   cout << "Whether float* objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<float*>::is_specialized
        << endl;
   cout << "Whether int objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<int>::is_specialized
        << endl;
   cout << "Whether int* objects have an explicit "
        << "specialization in the class: "
        << numeric_limits<int*>::is_specialized
        << endl;
}
```

Output

```
Whether float objects have an explicit specialization in the class: 1
Whether float* objects have an explicit specialization in the class: 0
Whether int objects have an explicit specialization in the class: 1
Whether int* objects have an explicit specialization in the class: 0
```

lowest

Returns the most negative finite value.

C++

```
static constexpr Type lowest() throw();
```

Return Value

Returns the most negative finite value.

Remarks

Returns the most negative finite value for the type (which is typically `min()` for integer types and `-max()` for floating-point types). The return value is meaningful if `is_bounded` is `true`.

max

Returns the maximum finite value for a type.

C++

```
static constexpr Type max() throw();
```

Return Value

The maximum finite value for a type.

Remarks

The maximum finite value is INT_MAX for type `int` and FLT_MAX for type `float`. The return value is meaningful if is_bounded is `true`.

Example

C++

```
// numeric_limits_max.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main() {
   cout << "The maximum value for type float is:  "
        << numeric_limits<float>::max( )
```

```
        << endl;
    cout << "The maximum value for type double is:  "
         << numeric_limits<double>::max( )
         << endl;
    cout << "The maximum value for type int is:  "
         << numeric_limits<int>::max( )
         << endl;
    cout << "The maximum value for type short int is:  "
         << numeric_limits<short int>::max( )
         << endl;
}
```

max_digits10

Returns the number of decimal digits required to make sure that two distinct values of the type have distinct decimal representations.

C++

```
static constexpr int max_digits10 = 0;
```

Return Value

Returns the number of decimal digits that are required to make sure that two distinct values of the type have distinct decimal representations.

Remarks

The member stores the number of decimal digits required to make sure that two distinct values of the type have distinct decimal representations.

max_exponent

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

C++

```
static constexpr int max_exponent = 0;
```

Return Value

The maximum integral radix-based exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The `max_exponent` is the value FLT_MAX_EXP for type **`float`**.

Example

C++

```
// numeric_limits_max_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum radix-based exponent for type float is:  "
        << numeric_limits<float>::max_exponent
        << endl;
   cout << "The maximum radix-based exponent for type double is:  "
        << numeric_limits<double>::max_exponent
        << endl;
   cout << "The maximum radix-based exponent for type long double is:  "
        << numeric_limits<long double>::max_exponent
        << endl;
}
```

Output

```
The maximum radix-based exponent for type float is:  128
The maximum radix-based exponent for type double is:  1024
The maximum radix-based exponent for type long double is:  1024
```

max_exponent10

Returns the maximum positive integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

C++

```
static constexpr int max_exponent10 = 0;
```

Return Value

The maximum integral base 10 exponent representable by the type.

Remarks

The member function return is meaningful only for floating-point types. The `max_exponent` is the value FLT_MAX_10 for type **`float`**.

Example

C++

```
// numeric_limits_max_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum base 10 exponent for type float is:  "
           << numeric_limits<float>::max_exponent10
           << endl;
   cout << "The maximum base 10 exponent for type double is:  "
           << numeric_limits<double>::max_exponent10
           << endl;
   cout << "The maximum base 10 exponent for type long double is:  "
           << numeric_limits<long double>::max_exponent10
           << endl;
}
```

Output

```
The maximum base 10 exponent for type float is:  38
The maximum base 10 exponent for type double is:  308
The maximum base 10 exponent for type long double is:  308
```

min

Returns the minimum normalized value for a type.

C++

```
static constexpr Type min() throw();
```

Return Value

The minimum normalized value for the type.

Remarks

The minimum normalized value is INT_MIN for type `int` and FLT_MIN for type `float`. The return value is meaningful if is_bounded is `true` or if is_signed is `false`.

Example

C++

```
// numeric_limits_min.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum value for type float is:  "
        << numeric_limits<float>::min( )
        << endl;
   cout << "The minimum value for type double is:  "
        << numeric_limits<double>::min( )
        << endl;
   cout << "The minimum value for type int is:  "
        << numeric_limits<int>::min( )
        << endl;
   cout << "The minimum value for type short int is:  "
        << numeric_limits<short int>::min( )
        << endl;
}
```

Output

```
The minimum value for type float is:  1.17549e-038
The minimum value for type double is:  2.22507e-308
The minimum value for type int is:  -2147483648
The minimum value for type short int is:  -32768
```

min_exponent

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of radix is raised to that power.

C++

```
static constexpr int min_exponent = 0;
```

Return Value

The minimum integral radix-based exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The `min_exponent` is the value FLT_MIN_EXP for type **`float`**.

Example

C++

```
// numeric_limits_min_exponent.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum radix-based exponent for type float is:  "
        << numeric_limits<float>::min_exponent
        << endl;
   cout << "The minimum radix-based exponent for type double is:  "
        << numeric_limits<double>::min_exponent
        << endl;
   cout << "The minimum radix-based exponent for type long double is:  "
         << numeric_limits<long double>::min_exponent
        << endl;
}
```

Output

```
The minimum radix-based exponent for type float is:  -125
The minimum radix-based exponent for type double is:  -1021
The minimum radix-based exponent for type long double is:  -1021
```

min_exponent10

Returns the maximum negative integral exponent that the floating-point type can represent as a finite value when a base of ten is raised to that power.

C++

```
static constexpr int min_exponent10 = 0;
```

Return Value

The minimum integral base 10 exponent representable by the type.

Remarks

The member function is meaningful only for floating-point types. The `min_exponent10` is the value FLT_MIN_10_EXP for type **`float`**.

Example

C++

```
// numeric_limits_min_exponent10.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The minimum base 10 exponent for type float is:  "
        << numeric_limits<float>::min_exponent10
        << endl;
   cout << "The minimum base 10 exponent for type double is:  "
        << numeric_limits<double>::min_exponent10
        << endl;
   cout << "The minimum base 10 exponent for type long double is:  "
        << numeric_limits<long double>::min_exponent10
        << endl;
}
```

Output

```
The minimum base 10 exponent for type float is:  -37
The minimum base 10 exponent for type double is:  -307
```

```
The minimum base 10 exponent for type long double is:  -307
```

quiet_NaN

Returns the representation of a quiet not a number (NAN) for the type.

C++

```
static constexpr Type quiet_NaN() throw();
```

Return Value

The representation of a quiet NAN for the type.

Remarks

The return value is meaningful only if has_quiet_NaN is `true`.

Example

C++

```
// numeric_limits_quiet_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The quiet NaN for type float is:  "
        << numeric_limits<float>::quiet_NaN( )
        << endl;
   cout << "The quiet NaN for type int is:  "
        << numeric_limits<int>::quiet_NaN( )
        << endl;
   cout << "The quiet NaN for type long double is:  "
        << numeric_limits<long double>::quiet_NaN( )
        << endl;
}
```

Output

```
The quiet NaN for type float is:  1.#QNAN
The quiet NaN for type int is:  0
The quiet NaN for type long double is:  1.#QNAN
```

radix

Returns the integral base, referred to as radix, used for the representation of a type.

C++

```
static constexpr int radix = 0;
```

Return Value

The integral base for the representation of the type.

Remarks

The base is 2 for the predefined integer types, and the base to which the exponent is raised, or FLT_RADIX, for the predefined floating-point types.

Example

C++

```
// numeric_limits_radix.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The base for type float is:  "
        << numeric_limits<float>::radix
        << endl;
   cout << "The base for type int is:  "
        << numeric_limits<int>::radix
        << endl;
   cout << "The base for type long double is:  "
        << numeric_limits<long double>::radix
        << endl;
}
```

Output

```
The base for type float is:  2
The base for type int is:  2
The base for type long double is:  2
```

round_error

Returns the maximum rounding error for the type.

C++

```
static constexpr Type round_error() throw();
```

Return Value

The maximum rounding error for the type.

Example

C++

```
// numeric_limits_round_error.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The maximum rounding error for type float is:  "
        << numeric_limits<float>::round_error( )
        << endl;
   cout << "The maximum rounding error for type int is:  "
        << numeric_limits<int>::round_error( )
        << endl;
   cout << "The maximum rounding error for type long double is:  "
        << numeric_limits<long double>::round_error( )
        << endl;
}
```

Output

```
The maximum rounding error for type float is:  0.5
The maximum rounding error for type int is:  0
```

```
The maximum rounding error for type long double is:  0.5
```

round_style

Returns a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

C++

```
static constexpr float_round_style round_style = round_toward_zero;
```

Return Value

A value from the `float_round_style` enumeration that describes the rounding style.

Remarks

The member stores a value that describes the various methods that an implementation can choose for rounding a floating-point value to an integer value.

The round style is hard coded in this implementation, so even if the program starts up with a different rounding mode, that value will not change.

Example

C++

```
// numeric_limits_round_style.cpp
// compile with: /EHsc
#include <iostream>
#include <float.h>
#include <limits>

using namespace std;

int main( )
{
   cout << "The rounding style for a double type is: "
        << numeric_limits<double>::round_style << endl;
   _controlfp_s(NULL,_RC_DOWN,_MCW_RC );
   cout << "The rounding style for a double type is now: "
        << numeric_limits<double>::round_style << endl;
   cout << "The rounding style for an int type is: "
        << numeric_limits<int>::round_style << endl;
}
```

Output

```
The rounding style for a double type is: 1
The rounding style for a double type is now: 1
The rounding style for an int type is: 0
```

signaling_NaN

Returns the representation of a signaling not a number (NAN) for the type.

C++

```
static constexpr Type signaling_NaN() throw();
```

Return Value

The representation of a signaling NAN for the type.

Remarks

The return value is meaningful only if has_signaling_NaN is `true`.

Example

C++

```
// numeric_limits_signaling_nan.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "The signaling NaN for type float is:  "
        << numeric_limits<float>::signaling_NaN( )
        << endl;
   cout << "The signaling NaN for type int is:  "
        << numeric_limits<int>::signaling_NaN( )
        << endl;
   cout << "The signaling NaN for type long double is:  "
        << numeric_limits<long double>::signaling_NaN( )
```

```
        << endl;
}
```

tinyness_before

Tests whether a type can determine that a value is too small to represent as a normalized value before rounding it.

C++

```
static constexpr bool tinyness_before = false;
```

Return Value

`true` if the type can detect tiny values before rounding; `false` if it cannot.

Remarks

Types that can detect tinyness were included as an option with IEC 559 floating-point representations and its implementation can affect some results.

Example

C++

```
// numeric_limits_tinyness_before.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float types can detect tinyness before rounding: "
        << numeric_limits<float>::tinyness_before
        << endl;
   cout << "Whether double types can detect tinyness before rounding: "
        << numeric_limits<double>::tinyness_before
        << endl;
   cout << "Whether long int types can detect tinyness before rounding: "
        << numeric_limits<long int>::tinyness_before
        << endl;
   cout << "Whether unsigned char types can detect tinyness before rounding:
"
```

```
        << numeric_limits<unsigned char>::tinyness_before
        << endl;
}
```

Output

```
Whether float types can detect tinyness before rounding: 1
Whether double types can detect tinyness before rounding: 1
Whether long int types can detect tinyness before rounding: 0
Whether unsigned char types can detect tinyness before rounding: 0
```

traps

Tests whether trapping that reports on arithmetic exceptions is implemented for a type.

C++

```
static constexpr bool traps = false;
```

Return Value

`true` if trapping is implemented for the type; `false` if it is not.

Example

C++

```
// numeric_limits_traps.cpp
// compile with: /EHsc
#include <iostream>
#include <limits>

using namespace std;

int main( )
{
   cout << "Whether float types have implemented trapping: "
        << numeric_limits<float>::traps
        << endl;
   cout << "Whether double types have implemented trapping: "
        << numeric_limits<double>::traps
        << endl;
   cout << "Whether long int types have implemented trapping: "
        << numeric_limits<long int>::traps
        << endl;
   cout << "Whether unsigned char types have implemented trapping: "
```

```
        << numeric_limits<unsigned char>::traps
        << endl;
}
```

Output

```
Whether float types have implemented trapping: 1
Whether double types have implemented trapping: 1
Whether long int types have implemented trapping: 0
Whether unsigned char types have implemented trapping: 0
```

<list>

Article • 12/07/2021

Defines the container class template list and several supporting templates.

Syntax

```
C++
```

```
#include <list>
```

> **Note**
>
> The <list> library also uses the `#include <initializer_list>` statement.

Members

Operators

Name	Description
operator!=	Tests if the list object on the left side of the operator is not equal to the list object on the right side.
operator<	Tests if the list object on the left side of the operator is less than the list object on the right side.
operator<=	Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.
operator==	Tests if the list object on the left side of the operator is equal to the list object on the right side.
operator>	Tests if the list object on the left side of the operator is greater than the list object on the right side.
operator>=	Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

Functions

Name	Description
swap	Exchanges the elements of two lists.

Classes

Name	Description
list Class	A class template of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<list>` operators

Article • 12/07/2021

operator!=

Tests if the list object on the left side of the operator is not equal to the list object on the right side.

C++

```
bool operator!=(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left
An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the lists are not equal; `false` if the lists are equal.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// list_op_ne.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
```

```
int main( )
{
using namespace std;
list <int> c1, c2;
c1.push_back( 1 );
c2.push_back( 2 );

if ( c1 != c2 )
cout << "Lists not equal." << endl;
else
cout << "Lists equal." << endl;
}
/* Output:
Lists not equal.
*/
```

`operator<`

Tests if the list object on the left side of the operator is less than the list object on the right side.

C++

```
bool operator<(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left
An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise `false`.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// list_op_lt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 < c2 )
      cout << "List c1 is less than list c2." << endl;
   else
      cout << "List c1 is not less than list c2." << endl;
}
/* Output:
List c1 is less than list c2.
*/
```

`operator<=`

Tests if the list object on the left side of the operator is less than or equal to the list object on the right side.

C++

```
bool operator<=(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left
An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise `false`.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// list_op_le.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 4 );

   c2.push_back( 1 );
   c2.push_back( 3 );

   if ( c1 <= c2 )
      cout << "List c1 is less than or equal to list c2." << endl;
   else
      cout << "List c1 is greater than list c2." << endl;
}
/* Output:
List c1 is less than or equal to list c2.
*/
```

operator==

Tests if the list object on the left side of the operator is equal to the list object on the right side.

```
C++
```

```
bool operator==(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left
An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the list on the left side of the operator is equal to the list on the right side of the operator; otherwise `false`.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. Two lists are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// list_op_eq.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
   using namespace std;

   list <int> c1, c2;
```

```
    c1.push_back( 1 );
    c2.push_back( 1 );

    if ( c1 == c2 )
       cout << "The lists are equal." << endl;
    else
       cout << "The lists are not equal." << endl;
}
/* Output:
The lists are equal.
*/
```

`operator>`

Tests if the list object on the left side of the operator is greater than the list object on the right side.

C++

```
bool operator>(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left
An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// list_op_gt.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 > c2 )
      cout << "List c1 is greater than list c2." << endl;
   else
      cout << "List c1 is not greater than list c2." << endl;
}
/* Output:
List c1 is greater than list c2.
*/
```

`operator>=`

Tests if the list object on the left side of the operator is greater than or equal to the list object on the right side.

C++

```
bool operator>=(
    const list<Type, Allocator>& left,
    const list<Type, Allocator>& right);
```

Parameters

left

An object of type `list`.

right
An object of type `list`.

Return Value

`true` if the list on the left side of the operator is greater than or equal to the list on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between list objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// list_op_ge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2;
   c1.push_back( 1 );
   c1.push_back( 3 );
   c1.push_back( 1 );

   c2.push_back( 1 );
   c2.push_back( 2 );
   c2.push_back( 2 );

   if ( c1 >= c2 )
      cout << "List c1 is greater than or equal to list c2." << endl;
   else
      cout << "List c1 is less than list c2." << endl;
}
/* Output:
List c1 is greater than or equal to list c2.
*/
```

`list` Class

Article • 08/03/2021

The C++ Standard Library list class is a class template of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type *`Type`*.

Syntax

```
C++
```

```
template <class Type, class Allocator= allocator<Type>>
class list
```

Parameters

`Type`

The element data type to be stored in the list.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the list's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Type>`.

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the class deque container is superior when random access is needed and insertions and deletions at both the beginning and the end of a sequence are at a premium.

The list member functions merge, reverse, unique, remove, and remove_if have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid.

Include the C++ Standard Library standard header `<list>` to define the container class template list and several supporting templates.

Members

Constructors

Name	Description
list	Constructs a list of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other list.

Typedefs

Name	Description
allocator_type	A type that represents the `allocator` class for a list object.
const_iterator	A type that provides a bidirectional iterator that can read a `const` element in a list.
const_pointer	A type that provides a pointer to a `const` element in a list.
const_reference	A type that provides a reference to a `const` element stored in a list for reading and performing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in a list.
difference_type	A type that provides the difference between two iterators that refer to elements within the same list.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a list.
pointer	A type that provides a pointer to an element in a list.
reference	A type that provides a reference to a `const` element stored in a list for reading and performing `const` operations.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

Name	Description
size_type	A type that counts the number of elements in a list.
value_type	A type that represents the data type stored in a list.

Functions

Name	Description
assign	Erases elements from a list and copies a new set of elements to the target list.
back	Returns a reference to the last element of a list.
begin	Returns an iterator addressing the first element in a list.
cbegin	Returns a const iterator addressing the first element in a list.
cend	Returns a const iterator that addresses the location succeeding the last element in a list.
clear	Erases all the elements of a list.
crbegin	Returns a const iterator addressing the first element in a reversed list.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed list.
emplace	Inserts an element constructed in place into a list at a specified position.
emplace_back	Adds an element constructed in place to the end of a list.
emplace_front	Adds an element constructed in place to the beginning of a list.
empty	Tests if a list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a list.
erase	Removes an element or a range of elements in a list from specified positions.
front	Returns a reference to the first element in a list.
get_allocator	Returns a copy of the `allocator` object used to construct a list.
insert	Inserts an element or a number of elements or a range of elements into a list at a specified position.
max_size	Returns the maximum length of a list.

Name	Description
merge	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_back	Deletes the element at the end of a list.
pop_front	Deletes the element at the beginning of a list.
push_back	Adds an element to the end of a list.
push_front	Adds an element to the beginning of a list.
rbegin	Returns an iterator addressing the first element in a reversed list.
remove	Erases elements in a list that match a specified value.
remove_if	Erases elements from the list for which a specified predicate is satisfied.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed list.
resize	Specifies a new size for a list.
reverse	Reverses the order in which the elements occur in a list.
size	Returns the number of elements in a list.
sort	Arranges the elements of a list in ascending order or with respect to some other order relation.
splice	Removes elements from the argument list and inserts them into the target list.
swap	Exchanges the elements of two lists.
unique	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.

Operators

Name	Description
operator=	Replaces the elements of the list with a copy of another list.

Requirements

Header: `<list>`

`allocator_type`

A type that represents the allocator class for a list object.

```
C++
```

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter *`Allocator`*.

Example

See the example for get_allocator.

`assign`

Erases elements from a list and copies a new set of elements to a target list.

```
C++
```

```
void assign(
    size_type Count,
    const Type& Val);

void assign
    initializer_list<Type> IList);

template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);
```

Parameters

`First`
Position of the first element in the range of elements to be copied from the argument list.

`Last`
Position of the first element just beyond the range of elements to be copied from the argument list.

`Count`

The number of copies of an element being inserted into the list.

`Val`

The value of the element being inserted into the list.

`IList`

The initializer_list that contains the elements to be inserted.

Remarks

After erasing any existing elements in the target list, assign either inserts a specified range of elements from the original list or from some other list into the target list or inserts copies of a new element of a specified value into the target list

Example

C++

```
// list_assign.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    list<int> c1, c2;
    list<int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign(7, 4);
```

```
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign({ 10, 20, 30, 40 });
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 10 20 30c1 = 50 60c1 = 4 4 4 4 4 4 4c1 = 10 20 30 40
```

back

Returns a reference to the last element of a list.

C++

```
reference back();

const_reference back() const;
```

Return Value

The last element of the list. If the list is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the list object cannot be modified. If the return value of `back` is assigned to a `reference`, the list object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty list. See Checked Iterators for more information.

Example

C++

```cpp
// list_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

begin

Returns an iterator addressing the first element in a list.

C++

```cpp
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the list or to the location succeeding an empty list.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the list object cannot be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the list object can be modified.

Example

C++

```
// list_begin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::const_iterator c1_cIter;

   c1.push_back( 1 );
   c1.push_back( 2 );

   c1_Iter = c1.begin( );
   cout << "The first element of c1 is " << *c1_Iter << endl;

*c1_Iter = 20;
   c1_Iter = c1.begin( );
   cout << "The first element of c1 is now " << *c1_Iter << endl;

   // The following line would be an error because iterator is const
   // *c1_cIter = 200;
}
```

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

cbegin

Returns a **`const`** iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++
```

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

clear

Erases all the elements of a list.

```
C++
```

```
void clear();
```

Example

```
C++
```

```
// list_clear.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "The size of the list is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of list after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the list is initially 3
The size of list after clearing is 0
```

`const_iterator`

A type that provides a bidirectional iterator that can read a `const` element in a list.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for back.

`const_pointer`

Provides a pointer to a `const` element in a list.

C++

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a list object.

`const_reference`

A type that provides a reference to a `const` element stored in a list for reading and performing `const` operations.

C++

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

Example

C++

```
// list_const_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   const list <int> c2 = c1;
   const int &i = c2.front( );
   const int &j = c2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error because c2 is const
   // c2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

`const_reverse_iterator`

A type that provides a bidirectional iterator that can read any **`const`** element in a list.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the list in reverse.

Example

See the example for rbegin.

`crbegin`

Returns a const iterator addressing the first element in a reversed list.

C++

```
const_reverse_iterator rbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed `list`).

Remarks

`crbegin` is used with a reversed list just as list::begin is used with a `list`.

With the return value of `crbegin`, the list object cannot be modified. list::rbegin can be used to iterate through a list backwards.

Example

C++

```
// list_crbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_crIter = c1.crbegin( );
   cout << "The last element in the list is " << *c1_crIter << "." << endl;
}
```

Output

```
The last element in the list is 30.
```

crend

Returns a const iterator that addresses the location succeeding the last element in a reversed list.

C++

```
const_reverse_iterator rend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed `list`).

Remarks

`crend` is used with a reversed list just as list::end is used with a `list`.

With the return value of `crend`, the `list` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `list`.

The value returned by `crend` should not be dereferenced.

Example

C++

```
// list_crend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_crIter = c1.crend( );
   c1_crIter --;  // Decrementing a reverse iterator moves it forward in
                  // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_crIter << endl;
}
```

Output

```
The first element in the list is: 10
```

`difference_type`

A signed integer type that can be used to represent the number of elements of a list in a range between elements pointed to by iterators.

C++

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container, such as vector Class.

Example

C++

```
// list_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <list>
#include <algorithm>

int main( )
{
   using namespace std;

   list <int> c1;
   list <int>::iterator   c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

    list <int>::difference_type df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

`emplace`

Inserts an element constructed in place into a list at a specified position.

C++

```
void emplace(iterator Where, Type&& val);
```

Parameters

`Where`

The position in the target list where the first element is inserted.

`val`

The element added to the end of the `list`.

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

C++

```
// list_emplace.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace(c2.begin(), move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Moved first element: a
```

`emplace_back`

Adds an element constructed in place to the end of a list.

C++

```
void emplace_back(Type&& val);
```

Parameters

`val`

The element added to the end of the list.

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

C++

```
// list_emplace_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace_back( move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Moved first element: a
```

`emplace_front`

Adds an element constructed in place to the beginning of a list.

```
C++
```

```
void emplace_front(Type&& val);
```

Parameters

val

The element added to the beginning of the list.

Remarks

If an exception is thrown, the `list` is left unaltered and the exception is rethrown.

Example

```
C++
```

```
// list_emplace_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace_front( move( str ) );
   cout << "Moved first element: " << c2.front( ) << endl;
}
```

```
Output
```

```
Moved first element: a
```

`empty`

Tests if a list is empty.

```
```

C++

```
bool empty() const;
```

Return Value

`true` if the list is empty; `false` if the list is not empty.

Example

C++

```
// list_empty.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The list is empty." << endl;
   else
      cout << "The list is not empty." << endl;
}
```

Output

```
The list is not empty.
```

end

Returns an iterator that addresses the location succeeding the last element in a list.

C++

```
const_iterator end() const;
iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a list. If the list is empty, then `list::end == list::begin`.

Remarks

`end` is used to test whether an iterator has reached the end of its list.

Example

C++

```
// list_end.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_Iter = c1.end( );
   c1_Iter--;
   cout << "The last integer of c1 is " << *c1_Iter << endl;

   c1_Iter--;
*c1_Iter = 400;
   cout << "The new next-to-last integer of c1 is "
        << *c1_Iter << endl;

   // If a const iterator had been declared instead with the line:
   // list <int>::const_iterator c1_Iter;
   // an error would have resulted when inserting the 400

   cout << "The list is now:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The list is now: 10 400 30
```

`erase`

Removes an element or a range of elements in a list from specified positions.

```
C++
```

```
iterator erase(iterator Where);
iterator erase(iterator first, iterator last);
```

Parameters

`Where`

Position of the element to be removed from the list.

`first`

Position of the first element removed from the list.

`last`

Position just beyond the last element removed from the list.

Return Value

A bidirectional iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the list if no such element exists.

Remarks

No reallocation occurs, so iterators and references become invalid only for the erased elements.

`erase` never throws an exception.

Example

```
C++
```

```
// list_erase.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial list is:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the list becomes:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
   cout << "After erasing all elements but the first, the list becomes: ";
   for (Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

Output

```
The initial list is: 10 20 30 40 50
After erasing the first element, the list becomes: 20 30 40 50
After erasing all elements but the first, the list becomes:  20
```

`front`

Returns a reference to the first element in a list.

C++

```
reference front();
const_reference front() const;
```

Return Value

If the list is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the list object cannot be modified. If the return value of `front` is assigned to a `reference`, the list object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty list. See Checked Iterators for more information.

Example

C++

```
// list_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main() {
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );

   int& i = c1.front();
   const int& ii = c1.front();

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The first integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The first integer of c1 is 11
```

`get_allocator`

Returns a copy of the allocator object used to construct a list.

C++

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the list.

Remarks

Allocators for the list class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// list_get_allocator.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   list <int> c1;
   list <int, allocator<int> > c2 = list <int, allocator<int> >(
allocator<int>( ) );

   // c3 will use the same allocator class as c1
   list <int> c3( c1.get_allocator( ) );

   list<int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

`insert`

Inserts an element or a number of elements or a range of elements into a list at a specified position.

C++

```
iterator insert(iterator Where, const Type& Val);
iterator insert(iterator Where, Type&& Val);
```

```
void insert(iterator Where, size_type Count, const Type& Val);
iterator insert(iterator Where, initializer_list<Type> IList);

template <class InputIterator>
void insert(iterator Where, InputIterator First, InputIterator Last);
```

Parameters

Where

The position in the target list where the first element is inserted.

Val

The value of the element being inserted into the list.

Count

The number of elements being inserted into the list.

First

The position of the first element in the range of elements in the argument list to be copied.

Last

The position of the first element beyond the range of elements in the argument list to be copied.

Return Value

The first two insert functions return an iterator that points to the position where the new element was inserted into the list.

Example

C++

```
// list_class_insert.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    list <int> c1, c2;
    list <int>::iterator Iter;
```

```
    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    c1.insert(Iter, 100);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    Iter++;
    c1.insert(Iter, 2, 200);

    cout << "c1 =";
    for(auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.insert(++c1.begin(), c2.begin(), --c2.end());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    // initialize a list of strings by moving
    list < string > c3;
    string str("a");

    c3.insert(c3.begin(), move(str));
    cout << "Moved first element: " << c3.front() << endl;

    // Assign with an initializer_list
    list <int> c4{ {1, 2, 3, 4} };
    c4.insert(c4.begin(), { 5, 6, 7, 8 });

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;
}
```

`iterator`

A type that provides a bidirectional iterator that can read or modify any element in a list.

```
C++
```

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin.

`list`

Constructs a list of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other list.

```
C++
```

```
list();
explicit list(const Allocator& Al);
explicit list(size_type Count);
list(size_type Count, const Type& Val);
list(size_type Count, const Type& Val, const Allocator& Al);

list(const list& Right);
list(list&& Right);
list(initializer_list<Type> IList, const Allocator& Al);

template <class InputIterator>
list(InputIterator First, InputIterator Last);

template <class InputIterator>
list(InputIterator First, InputIterator Last, const Allocator& Al);
```

Parameters

`Al`

The allocator class to use with this object.

Count

The number of elements in the list constructed.

Val

The value of the elements in the list.

Right

The list of which the constructed list is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The initializer_list that contains the elements to be copied.

Remarks

All constructors store an allocator object (*Al*) and initialize the list.

get_allocator returns a copy of the allocator object used to construct a list.

The first two constructors specify an empty initial list, the second specifying the allocator type (*Al*) to be used.

The third constructor specifies a repetition of a specified number (*Count*) of elements of the default value for class `Type`.

The fourth and fifth constructors specify a repetition of (*Count*) elements of value *Val*.

The sixth constructor specifies a copy of the list *Right*.

The seventh constructor moves the list *Right*.

The eighth constructor uses an initializer_list to specify the elements.

The next two constructors copy the range `[First, Last)` of a list.

None of the constructors perform any interim reallocations.

Example

```
```

C++

```cpp
// list_class_list.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    // Create an empty list c0
    list <int> c0;

    // Create a list c1 with 3 elements of default value 0
    list <int> c1(3);

    // Create a list c2 with 5 elements of value 2
    list <int> c2(5, 2);

    // Create a list c3 with 3 elements of value 1 and with the
    // allocator of list c2
    list <int> c3(3, 1, c2.get_allocator());

    // Create a copy, list c4, of list c2
    list <int> c4(c2);

    // Create a list c5 by copying the range c4[ first,  last)
    list <int>::iterator c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    list <int> c5(c4.begin(), c4_Iter);

    // Create a list c6 by copying the range c4[ first,  last) and with
    // the allocator of list c2
    c4_Iter = c4.begin();
    c4_Iter++;
    c4_Iter++;
    c4_Iter++;
    list <int> c6(c4.begin(), c4_Iter, c2.get_allocator());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    cout << "c2 =";
    for (auto c : c2)
        cout << " " << c;
    cout << endl;

    cout << "c3 =";
    for (auto c : c3)
        cout << " " << c;
    cout << endl;
```

```
    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;

    cout << "c5 =";
    for (auto c : c5)
        cout << " " << c;
    cout << endl;

    cout << "c6 =";
    for (auto c : c6)
        cout << " " << c;
    cout << endl;

    // Move list c6 to list c7
    list <int> c7(move(c6));
    cout << "c7 =";
    for (auto c : c7)
        cout << " " << c;
    cout << endl;

    // Construct with initializer_list
    list<int> c8({ 1, 2, 3, 4 });
    cout << "c8 =";
    for (auto c : c8)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 0 0 0c2 = 2 2 2 2 2c3 = 1 1 1c4 = 2 2 2 2 2c5 = 2 2c6 = 2 2 2c7 = 2 2
2c8 = 1 2 3 4
```

`max_size`

Returns the maximum length of a list.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the list.

Example

C++

```
// list_max_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   i = c1.max_size( );
   cout << "Maximum possible length of the list is " << i << "." << endl;
}
```

`merge`

Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.

C++

```
void merge(list<Type, Allocator>& right);

template <class Traits>
void merge(list<Type, Allocator>& right, Traits comp);
```

Parameters

`right`

The argument list to be merged with the target list.

`comp`

The comparison operator used to order the elements of the target list.

Remarks

The argument list *`right`* is merged with the target list.

Both argument and target lists must be ordered with the same comparison relation by which the resulting sequence is to be ordered. The default order for the first member function is ascending order. The second member function imposes the user-specified comparison operation *comp* of class `Traits`.

Example

C++

```
// list_merge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter;

   c1.push_back( 3 );
   c1.push_back( 6 );
   c2.push_back( 2 );
   c2.push_back( 4 );
   c3.push_back( 5 );
   c3.push_back( 1 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   cout << "c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   c2.merge( c1 );  // Merge c1 into c2 in (default) ascending order
   c2.sort( greater<int>( ) );
   cout << "After merging c1 with c2 and sorting with >: c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   cout << "c3 =";
   for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
      cout << " " << *c3_Iter;
   cout << endl;

   c2.merge( c3, greater<int>( ) );
   cout << "After merging c3 with c2 according to the '>' comparison
relation: c2 =";
```

```
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
c1 = 3 6
c2 = 2 4
After merging c1 with c2 and sorting with >: c2 = 6 4 3 2
c3 = 5 1
After merging c3 with c2 according to the '>' comparison relation: c2 = 6 5
4 3 2 1
```

`operator=`

Replaces the elements of the list with a copy of another list.

C++

```
list& operator=(const list& right);
list& operator=(list&& right);
```

Parameters

`right`

The list being copied into the `list`.

Remarks

After erasing any existing elements in a `list`, the operator either copies or moves the contents of *`right`* into the `list`.

Example

C++

```
// list_operator_as.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list<int> v1, v2, v3;
   list<int>::iterator iter;

   v1.push_back(10);
   v1.push_back(20);
   v1.push_back(30);
   v1.push_back(40);
   v1.push_back(50);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = forward< list<int> >(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
}
```

`pointer`

Provides a pointer to an element in a list.

C++

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a list object.

`pop_back`

Deletes the element at the end of a list.

C++

```
void pop_back();
```

Remarks

The last element must not be empty. `pop_back` never throws an exception.

Example

C++

```
// list_pop_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the list, "
           "the last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the list, the last element is: 1
```

`pop_front`

Deletes the element at the beginning of a list.

C++

```
void pop_front();
```

Remarks

The first element must not be empty. `pop_front` never throws an exception.

Example

C++

```
// list_pop_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the list, "
        "the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the list, the first element
is: 2
```

push_back

Adds an element to the end of a list.

C++

```
void push_back(const Type& val);
void push_back(Type&& val);
```

Parameters

val

The element added to the end of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

C++

```
// list_push_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 1 );
   if ( c1.size( ) != 0 )
      cout << "Last element: " << c1.back( ) << endl;

   c1.push_back( 2 );
   if ( c1.size( ) != 0 )
      cout << "New last element: " << c1.back( ) << endl;

// move initialize a list of strings
   list <string> c2;
   string str("a");

   c2.push_back( move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved first element: a
```

`push_front`

Adds an element to the beginning of a list.

C++

```
void push_front(const Type& val);
void push_front(Type&& val);
```

Parameters

`val`

The element added to the beginning of the list.

Remarks

If an exception is thrown, the list is left unaltered and the exception is rethrown.

Example

C++

```
// list_push_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;

// move initialize a list of strings
   list <string> c2;
   string str("a");

   c2.push_front( move( str ) );
```

```
    cout << "Moved first element: " << c2.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
Moved first element: a
```

rbegin

Returns an iterator that addresses the first element in a reversed list.

C++

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed list).

Remarks

`rbegin` is used with a reversed list just as begin is used with a list.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the list object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the list object can be modified.

`rbegin` can be used to iterate through a list backwards.

Example

C++

```
// list_rbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   // If the following line replaced the line above, *c1_rIter = 40;
   // (below) would be an error
   //list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_rIter = c1.rbegin( );
   cout << "The last element in the list is " << *c1_rIter << "." << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration through a list in
   // reverse order
   cout << "The reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;

   c1_rIter = c1.rbegin( );
*c1_rIter = 40;
   cout << "The last element in the list is now " << *c1_rIter << "." <<
endl;
}
```

Output

```
The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.
```

reference

A type that provides a reference to an element stored in a list.

C++

```
typedef typename Allocator::reference reference;
```

Example

C++

```
// list_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

`remove`

Erases elements in a list that match a specified value.

C++

```
void remove(const Type& val);
```

Parameters

`val`

The value which, if held by an element, will result in that element's removal from the list.

Remarks

The order of the elements remaining is not affected.

Example

C++

```
// list_remove.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 5 );
   c1.push_back( 100 );
   c1.push_back( 5 );
   c1.push_back( 200 );
   c1.push_back( 5 );
   c1.push_back( 300 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove( 5 );
   cout << "After removing elements with value 5, the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 5 100 5 200 5 300
After removing elements with value 5, the list becomes c2 = 100 200 300
```

`remove_if`

Erases elements from a list for which a specified predicate is satisfied.

C++

```
template <class Predicate>
void remove_if(Predicate pred)
```

Parameters

pred

The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Example

C++

```
// list_remove_if.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

template <class T> class is_odd : public std::unary_function<T, bool>
{
public:
   bool operator( ) ( T& val )
   {
   return ( val % 2 ) == 1;
   }
};

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 3 );
   c1.push_back( 4 );
   c1.push_back( 5 );
   c1.push_back( 6 );
   c1.push_back( 7 );
   c1.push_back( 8 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.remove_if( is_odd<int>( ) );

   cout << "After removing the odd elements, "
        << "the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes c2 = 4 6 8
```

rend

Returns an iterator that addresses the location that follows the last element in a reversed list.

C++

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed list).

Remarks

`rend` is used with a reversed list just as end is used with a list.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the list object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, the list object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its list.

The value returned by `rend` should not be dereferenced.

Example

C++

```
// list_rend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error would
   // have resulted in the line modifying an element (commented below)
   // because the iterator would have been const
   // list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1_rIter = c1.rend( );
   c1_rIter --;  // Decrementing a reverse iterator moves it forward in
                 // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_rIter << endl;

   cout << "The list is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   // rend can be used to test if an iteration is through all of the
   // elements of a reversed list
   cout << "The reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;

   c1_rIter = c1.rend( );
   c1_rIter--;  // Decrementing the reverse iterator moves it backward
                // in the reversed list (to the last element here)

*c1_rIter = 40;  // This modification of the last element would have
                    // caused an error if a const_reverse iterator had
                    // been declared (as noted above)

   cout << "The modified reversed list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;
}
```

Output

```
The first element in the list is: 10
The list is: 10 20 30
The reversed list is: 30 20 10
The modified reversed list is: 30 20 40
```

`resize`

Specifies a new size for a list.

```
C++
```

```
void resize(size_type _Newsize);
void resize(size_type _Newsize, Type val);
```

Parameters

`_Newsize`

The new size of the list.

`val`

The value of the new elements to be added to the list if the new size is larger that the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the list's size is less than the requested size, *`_Newsize`*, elements are added to the list until it reaches the requested size.

If the list's size is larger than the requested size, the elements closest to the end of the list are deleted until the list reaches the size *`_Newsize`*.

If the present size of the list is the same as the requested size, no action is taken.

size reflects the current size of the list.

Example

```
C++
```

```
// list_resize.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
```

```
   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   c1.resize( 4,40 );
   cout << "The size of c1 is " << c1.size( ) << endl;
   cout << "The value of the last element is " << c1.back( ) << endl;

   c1.resize( 5 );
   cout << "The size of c1 is now " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;

   c1.resize( 2 );
   cout << "The reduced size of c1 is: " << c1.size( ) << endl;
   cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is 4
The value of the last element is 40
The size of c1 is now 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

reverse

Reverses the order in which the elements occur in a list.

C++

```
void reverse();
```

Example

C++

```
// list_reverse.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );

   cout << "c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.reverse( );
   cout << "Reversed c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
c1 = 10 20 30
Reversed c1 = 30 20 10
```

`reverse_iterator`

A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the list in reverse.

Example

See the example for rbegin.

`size`

Returns the number of elements in a list.

C++

```
size_type size() const;
```

Return Value

The current length of the list.

Example

C++

```
// list_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::size_type i;

   c1.push_back( 5 );
   i = c1.size( );
   cout << "List length is " << i << "." << endl;

   c1.push_back( 7 );
   i = c1.size( );
   cout << "List length is now " << i << "." << endl;
}
```

Output

```
List length is 1.
List length is now 2.
```

`size_type`

A type that counts the number of elements in a list.

C++

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for size.

`sort`

Arranges the elements of a list in ascending order or with respect to some other user-specified order.

C++

```
void sort();

template <class Traits>
    void sort(Traits comp);
```

Parameters

comp

The comparison operator used to order successive elements.

Remarks

The first member function puts the elements in ascending order by default.

The member template function orders the elements according to the user-specified comparison operation *comp* of class `Traits`.

Example

C++

```
// list_sort.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;

   c1.push_back( 20 );
```

```
   c1.push_back( 10 );
   c1.push_back( 30 );

   cout << "Before sorting: c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( );
   cout << "After sorting c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.sort( greater<int>( ) );
   cout << "After sorting with 'greater than' operation, c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

splice

Removes elements from a source list and inserts them into a destination list.

C++

```
// insert the entire source list
void splice(const_iterator Where, list<Type, Allocator>& Source);
void splice(const_iterator Where, list<Type, Allocator>&& Source);

// insert one element of the source list
void splice(const_iterator Where, list<Type, Allocator>& Source,
const_iterator Iter);
void splice(const_iterator Where, list<Type, Allocator>&& Source,
const_iterator Iter);

// insert a range of elements from the source list
void splice(const_iterator Where, list<Type, Allocator>& Source,
const_iterator First, const_iterator Last);
void splice(const_iterator Where, list<Type, Allocator>&& Source,
const_iterator First, const_iterator Last);
```

Parameters

Where

The position in the destination list before which to insert.

Source

The source list that is to be inserted into the destination list.

Iter

The element to be inserted from the source list.

First

The first element in the range to be inserted from the source list.

Last

The first position beyond the last element in the range to be inserted from the source list.

Remarks

The first pair of member functions inserts all elements in the source list into the destination list before the position referred to by *Where* and removes all elements from the source list. (`&Source` must not equal **`this`**.)

The second pair of member functions inserts the element referred to by *Iter* before the position in the destination list referred to by *Where* and removes *Iter* from the source list. (If `Where == Iter || Where == ++Iter`, no change occurs.)

The third pair of member functions inserts the range designated by [`First`, `Last`) before the element in the destination list referred to by *Where* and removes that range of elements from the source list. (If `&Source == this`, the range `[First, Last)` must not include the element pointed to by *Where*.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class iterator is incremented `N` times.

In all cases iterators, pointers, or references that refer to spliced elements remain valid and are transferred to the destination container.

Example

```
C++
```

```cpp
// list_splice.cpp
// compile with: /EHsc /W4
#include <list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    list<int> c1{10,11};
    list<int> c2{20,21,22};
    list<int> c3{30,31};
    list<int> c4{40,41,42,43};

    list<int>::iterator where_iter;
    list<int>::iterator first_iter;
    list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);

    first_iter = c3.begin();
    c2.splice(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);
```

```
    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    --last_iter;
    c2.splice(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}
```

Output

```
Beginning state of lists:c1 = 2 elements: (10) (11)c2 = 3 elements: (20)
(21) (22)c3 = 2 elements: (30) (31)c4 = 4 elements: (40) (41) (42) (43)After
splicing c1 into c2:c1 = 0 elements:c2 = 5 elements: (20) (10) (11) (21)
(22)After splicing the first element of c3 into c2:c3 = 1 elements: (31)c2 =
6 elements: (20) (10) (11) (30) (21) (22)After splicing a range of c4 into
c2:c4 = 2 elements: (40) (43)c2 = 8 elements: (20) (10) (11) (30) (41) (42)
(21) (22)
```

swap

Exchanges the elements of two lists.

C++

```
void swap(list<Type, Allocator>& right);
friend void swap(list<Type, Allocator>& left, list<Type, Allocator>& right)
```

Parameters

right

The list providing the elements to be swapped, or the list whose elements are to be exchanged with those of the list *left*.

left

A list whose elements are to be exchanged with those of the list *right*.

Example

C++

```cpp
// list_swap.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap( c1,c3 );

   cout << "After swapping with c3, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
The original list c1 is: 1 2 3
After swapping with c2, list c1 is: 10 20
After swapping with c3, list c1 is: 100
```

unique

Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from a list.

C++

```
void unique();

template <class BinaryPredicate>
void unique(BinaryPredicate pred);
```

Parameters

pred

The binary predicate used to compare successive elements.

Remarks

This function assumes that the list is sorted, so that all duplicate elements are adjacent. Duplicates that are not adjacent will not be deleted.

The first member function removes every element that compares equal to its preceding element.

The second member function removes every element that satisfies the predicate function *pred* when compared with its preceding element. You can use any of the binary function objects declared in the `<functional>` header for the argument *pred* or you can create your own.

Example

C++

```
// list_unique.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter,c3_Iter;
   not_equal_to<int> mypred;

   c1.push_back( -10 );
   c1.push_back( 10 );
   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 20 );
```

```
   c1.push_back( -10 );

   cout << "The initial list is c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   list <int> c2 = c1;
   c2.unique( );
   cout << "After removing successive duplicate elements, c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;

   list <int> c3 = c2;
   c3.unique( mypred );
   cout << "After removing successive unequal elements, c3 =";
   for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
      cout << " " << *c3_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

`value_type`

A type that represents the data type stored in a list.

C++

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter *`Type`*.

Example

C++

```
// list_value_type.cpp
// compile with: /EHsc
```

```
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

<locale>

Article • 12/07/2021

Defines class templates and functions that C++ programs can use to encapsulate and manipulate different cultural conventions regarding the representation and formatting of numeric, monetary, and calendric data, including internationalization support for character classification and string collation.

Syntax

```
C++

#include <locale>
```

Functions

Function	Description
has_facet	Tests if a particular facet is stored in a specified locale.
isalnum	Tests whether an element in a locale is an alphabetic or a numeric character.
isalpha	Tests whether an element in a locale is alphabetic character.
iscntrl	Tests whether an element in a locale is a control character.
isdigit	Tests whether an element in a locale is a numeric character.
isgraph	Tests whether an element in a locale is an alphanumeric or punctuation character.
islower	Tests whether an element in a locale is lower case.
isprint	Tests whether an element in a locale is a printable character.
ispunct	Tests whether an element in a locale is a punctuation character.
isspace	Tests whether an element in a locale is a whitespace character.
isupper	Tests whether an element in a locale is upper case.
isxdigit	Tests whether an element in a locale is a character used to represent a hexadecimal number.
tolower	Converts a character to lower case.
toupper	Converts a character to upper case.

Function	Description
use_facet	Returns a reference to a facet of a specified type stored in a locale.

Classes

Class	Description
codecvt	A class template that provides a facet used to convert between internal and external character encodings.
codecvt_base	A base class for the codecvt class that is used to define an enumeration type referred to as `result`, used as the return type for the facet member functions to indicate the result of a conversion.
codecvt_byname	A derived class template that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning conversions.
collate	A collate class template that provides a facet that handles string sorting conventions.
collate_byname	A derived class template that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning string sorting conventions.
ctype	A class template that provides a facet that is used to classify characters, convert from upper- and lowercase and between the native character set and that set used by the locale.
ctype<char>	A class that is an explicit specialization of class template `ctype<CharType>` to type `char`, describing an object that can serve as a locale facet to characterize various properties of a character of type `char`.
ctype_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
ctype_byname	A derived class template that describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.
locale	A class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.

Class	Description
messages	A class template that describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.
messages_base	A base class that describes an `int` type for the catalog of messages.
messages_byname	A derived class template that describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.
money_base	A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.
money_get	A class template that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to monetary values.
money_put	A class template that describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type **CharType**.
moneypunct	A class template that describes an object that can serve as a locale facet to describe the sequences of type **CharType** used to represent a monetary input field or a monetary output field.
moneypunct_byname	A derived class template that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting monetary input or output fields.
num_get	A class template that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to numeric values.
num_put	A class template that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type **CharType**.
numpunct	A class template that describes an object that can serve as a local facet to describe the sequences of type **CharType** used to represent information about the formatting and punctuation of numeric and Boolean expressions.
numpunct_byname	A derived class template that describes an object that can serve as a moneypunct facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.
time_base	A class that serves as a base class for facets of class template time_get, defining just the enumerated type dateorder and several constants of this type.

Class	Description
time_get	A class template that describes an object that can serve as a locale facet to control conversions of sequences of type **CharType** to time values.
time_get_byname	A derived class template that describes an object that can serve as a locale facet of type time_get<**CharType**, **InputIterator**>.
time_put	A class template that describes an object that can serve as a locale facet to control conversions of time values to sequences of type **CharType**.
time_put_byname	A derived class template that describes an object that can serve as a locale facet of type `time_put`<**CharType**, **OutputIterator**>.
wbuffer_convert Class	Describes a stream buffer that controls the transmission of elements to and from a byte stream buffer.
wstring_convert Class	A class template that performs conversions between a wide string and a byte string.

See also

Code Pages
Locale Names, Languages, and Country/Region Strings
Thread Safety in the C++ Standard Library

`<locale>` functions

Article • 12/07/2021

has_facet
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
tolower
toupper
use_facet

has_facet

Tests if a particular facet is stored in a specified locale.

C++

```
template <class Facet>
bool has_facet(const locale& Loc);
```

Parameters

Loc
The locale to be tested for the presence of a facet.

Return Value

`true` if the locale has the facet tested for; `false` if it does not.

Remarks

The template function is useful for checking whether nonmandatory facets are listed in a locale before `use_facet` is called to avoid the exception that would be thrown if it were not present.

Example

C++

```
// locale_has_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result = has_facet <ctype<char> > ( loc );
   cout << result << endl;
}
```

Output

```
1
```

isalnum

Tests whether an element in a locale is an alphabetic or a numeric character.

C++

```
template <class CharType>
bool isalnum(CharType Ch, const locale& Loc)
```

Parameters

Ch
The alphanumeric element to be tested.

Loc
The locale containing the alphanumeric element to be tested.

Return Value

`true` if the element tested is alphanumeric; `false` if it is not.

Example

C++

```
// locale_isalnum.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isalnum ( 'L', loc);
   bool result2 = isalnum ( '@', loc);
   bool result3 = isalnum ( '3', loc);

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not alphanumeric." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not alphanumeric." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "alphanumeric." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not alphanumeric." << endl;
}
```

Output

```
The character 'L' in the locale is alphanumeric.
The character '@' in the locale is  not alphanumeric.
The character '3' in the locale is alphanumeric.
```

isalpha

Tests whether an element in a locale is an alphabetic character.

```
C++
```

```
template <class CharType>
bool isalpha(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the alphabetic element to be tested.

Return Value

`true` if the element tested is alphabetic; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **alpha**, `Ch`).

Example

```
C++
```

```
// locale_isalpha.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isalpha ( 'L', loc);
   bool result2 = isalpha ( '@', loc);
   bool result3 = isalpha ( '3', loc);

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "alphabetic." << endl;
```

```
   else
      cout << "The character 'L' in the locale is "
           << " not alphabetic." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "alphabetic." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not alphabetic." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "alphabetic." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not alphabetic." << endl;
}
```

iscntrl

Tests whether an element in a locale is a control character.

C++

```
template <class CharType>
bool iscntrl(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is a control character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **cntrl**, `Ch`).

Example

C++

```
// locale_iscntrl.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = iscntrl ( 'L', loc );
   bool result2 = iscntrl ( '\n', loc );
   bool result3 = iscntrl ( '\t', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a control character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a control character." << endl;

   if ( result2 )
      cout << "The character-set 'backslash-n' in the locale\n is "
           << "a control character." << endl;
   else
      cout << "The character-set 'backslash-n' in the locale\n is "
           << " not a control character." << endl;

   if ( result3 )
      cout << "The character-set 'backslash-t' in the locale\n is "
           << "a control character." << endl;
   else
      cout << "The character-set 'backslash-n' in the locale \n is "
           << " not a control character." << endl;
}
```

isdigit

Tests whether an element in a locale is a numeric character.

C++

```
template <class CharType>
bool isdigit(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is a numeric character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **digit**, `Ch`).

Example

C++

```
// locale_is_digit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isdigit ( 'L', loc );
   bool result2 = isdigit ( '@', loc );
   bool result3 = isdigit ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a numeric character." << endl;

   if ( result2 )
      cout << "The character '@' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character '@' in the locale is "
           << " not a numeric character." << endl;
```

```
   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a numeric character." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not a numeric character." << endl;
}
```

isgraph

Tests whether an element in a locale is an alphanumeric or punctuation character.

C++

```
template <class CharType>
bool isgraph(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is an alphanumeric or a punctuation character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **graph**, `Ch`).

Example

C++

```
// locale_is_graph.cpp
// compile with: /EHsc
```

```
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isgraph ( 'L', loc );
   bool result2 = isgraph ( '\t', loc );
   bool result3 = isgraph ( '.', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character 'L' in the locale is\n "
           << " not an alphanumeric or punctuation character." << endl;

   if ( result2 )
      cout << "The character 'backslash-t' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character 'backslash-t' in the locale is\n "
           << "not an alphanumeric or punctuation character." << endl;

   if ( result3 )
      cout << "The character '.' in the locale is\n "
           << "an alphanumeric or punctuation character." << endl;
   else
      cout << "The character '.' in the locale is\n "
           << " not an alphanumeric or punctuation character." << endl;
}
```

islower

Tests whether an element in a locale is lower case.

C++

```
template <class CharType>
bool islower(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

`true` if the element tested is a lowercase character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **lower**, `Ch`).

Example

C++

```
// locale_islower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = islower ( 'L', loc );
   bool result2 = islower ( 'n', loc );
   bool result3 = islower ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a lowercase character." << endl;

   if ( result2 )
      cout << "The character 'n' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character 'n' in the locale is "
           << " not a lowercase character." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a lowercase character." << endl;
   else
      cout << "The character '3' in the locale is "
```

```
            << " not a lowercase character." << endl;
}
```

isprint

Tests whether an element in a locale is a printable character.

C++

```
template <class CharType>
bool isprint(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is a printable; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **print**, `Ch`).

Example

C++

```
// locale_isprint.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
```

```
    bool result1 = isprint ( 'L', loc );
    if ( result1 )
       cout << "The character 'L' in the locale is "
            << "a printable character." << endl;
    else
       cout << "The character 'L' in the locale is "
            << " not a printable character." << endl;

    bool result2 = isprint( '\t', loc );
    if ( result2 )
       cout << "The character 'backslash-t' in the locale is "
            << "a printable character." << endl;
    else
       cout << "The character 'backslash-t' in the locale is "
            << " not a printable character." << endl;

    bool result3 = isprint( '\n', loc );
    if ( result3 )
       cout << "The character 'backslash-n' in the locale is "
            << "a printable character." << endl;
    else
       cout << "The character 'backslash-n' in the locale is "
            << " not a printable character." << endl;
}
```

ispunct

Tests whether an element in a locale is a punctuation character.

C++

```
template <class CharType>
bool ispunct(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is a punctuation character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **punct**, `Ch`).

Example

C++

```
// locale_ispunct.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = ispunct ( 'L', loc );
   bool result2 = ispunct ( ';', loc );
   bool result3 = ispunct ( '*', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a punctuation character." << endl;

   if ( result2 )
      cout << "The character ';' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character ';' in the locale is "
           << " not a punctuation character." << endl;

   if ( result3 )
      cout << "The character '*' in the locale is "
           << "a punctuation character." << endl;
   else
      cout << "The character '*' in the locale is "
           << " not a punctuation character." << endl;
}
```

isspace

Tests whether an element in a locale is a whitespace character.

C++

```
template <class CharType>
bool isspace(CharType Ch, const locale& Loc)
```

Parameters

Ch

The element to be tested.

Loc

The locale containing the element to be tested.

Return Value

`true` if the element tested is a whitespace character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **space**, `Ch`).

Example

C++

```
// locale_isspace.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isspace ( 'L', loc );
   bool result2 = isspace ( '\n', loc );
   bool result3 = isspace ( ' ', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a whitespace character." << endl;
```

```
   if ( result2 )
      cout << "The character 'backslash-n' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character 'backslash-n' in the locale is "
           << " not a whitespace character." << endl;

   if ( result3 )
      cout << "The character ' ' in the locale is "
           << "a whitespace character." << endl;
   else
      cout << "The character ' ' in the locale is "
           << " not a whitespace character." << endl;
}
```

isupper

Tests whether an element in a locale is in upper case.

C++

```
template <class CharType>
bool isupper(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is an uppercase character; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **upper**, `Ch`).

Example

C++

```
// locale_isupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isupper ( 'L', loc );
   bool result2 = isupper ( 'n', loc );
   bool result3 = isupper ( '3', loc );

   if ( result1 )
      cout << "The character 'L' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character 'L' in the locale is "
           << " not a uppercase character." << endl;

   if ( result2 )
      cout << "The character 'n' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character 'n' in the locale is "
           << " not a uppercase character." << endl;

   if ( result3 )
      cout << "The character '3' in the locale is "
           << "a uppercase character." << endl;
   else
      cout << "The character '3' in the locale is "
           << " not a uppercase character." << endl;
}
```

isxdigit

Tests whether an element in a locale is a character used to represent a hexadecimal number.

C++

```
template <class CharType>
bool isxdigit(CharType Ch, const locale& Loc)
```

Parameters

Ch
The element to be tested.

Loc
The locale containing the element to be tested.

Return Value

`true` if the element tested is a character used to represent a hexadecimal number; `false` if it is not.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). is(**ctype**< **CharType**>:: **xdigit**, `Ch`).

Hexadecimal digits use base 16 to represent numbers, using the numbers 0 through 9 plus case-insensitive letters A through F to represent the decimal numbers 0 through 15.

Example

C++

```
// locale_isxdigit.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>

using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = isxdigit ( '5', loc );
   bool result2 = isxdigit ( 'd', loc );
   bool result3 = isxdigit ( 'q', loc );

   if ( result1 )
      cout << "The character '5' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character '5' in the locale is "
           << " not a hexidecimal digit-character." << endl;

   if ( result2 )
```

```
      cout << "The character 'd' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character 'd' in the locale is "
           << " not a hexidecimal digit-character." << endl;

   if ( result3 )
      cout << "The character 'q' in the locale is "
           << "a hexidecimal digit-character." << endl;
   else
      cout << "The character 'q' in the locale is "
           << " not a hexidecimal digit-character." << endl;
}
```

tolower

Converts a character to lower case.

```
C++
```

```
template <class CharType>
CharType tolower(CharType Ch, const locale& Loc)
```

Parameters

Ch
The character to be converted to lower case.

Loc
The locale containing the character to be converted.

Return Value

The character converted to lower case.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). tolower(`Ch`).

Example

```
C++
```

```
// locale_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   char result1 = tolower ( 'H', loc );
   cout << "The lower case of 'H' in the locale is: "
        << result1 << "." << endl;
   char result2 = tolower ( 'h', loc );
   cout << "The lower case of 'h' in the locale is: "
        << result2 << "." << endl;
   char result3 = tolower ( '$', loc );
   cout << "The lower case of '$' in the locale is: "
        << result3 << "." << endl;
}
```

toupper

Converts a character to upper case.

C++

```
template <class CharType>
CharType toupper(CharType Ch, const locale& Loc)
```

Parameters

Ch
The character to be converted to upper case.

Loc
The locale containing the character to be converted.

Return Value

The character converted to upper case.

Remarks

The template function returns use_facet< ctype< **CharType**> >(`Loc`). toupper(`Ch`).

Example

```
C++
```

```
// locale_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   char result1 = toupper ( 'h', loc );
   cout << "The upper case of 'h' in the locale is: "
        << result1 << "." << endl;
   char result2 = toupper ( 'H', loc );
   cout << "The upper case of 'H' in the locale is: "
        << result2 << "." << endl;
   char result3 = toupper ( '$', loc );
   cout << "The upper case of '$' in the locale is: "
        << result3 << "." << endl;
}
```

use_facet

Returns a reference to a facet of a specified type stored in a locale.

```
C++
```

```
template <class Facet>
const Facet& use_facet(const locale& Loc);
```

Parameters

Loc
The const locale containing the type of facet being referenced.

Return Value

A reference to the facet of class `Facet` contained within the argument locale.

Remarks

The reference to the facet returned by the template function remains valid as long as any copy of the containing locale exists. If no such facet object of class `Facet` is listed in the argument locale, the function throws a `bad_cast` exception.

Example

C++

```
// locale_use_facet.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );
   bool result1 = use_facet<ctype<char> > ( loc1 ).is(
   ctype_base::alpha, 'a'
);
   bool result2 = use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha,
'!'
   );

   if ( result1 )
      cout << "The character 'a' in locale loc1 is alphabetic."
           << endl;
   else
      cout << "The character 'a' in locale loc1 is not alphabetic."
           << endl;

   if ( result2 )
      cout << "The character '!' in locale loc2 is alphabetic."
           << endl;
   else
      cout << "The character '!' in locale loc2 is not alphabetic."
           << endl;
}
```

Output

```
The character 'a' in locale loc1 is alphabetic.
The character '!' in locale loc2 is not alphabetic.
```

See also

<locale>

`codecvt` Class

Article • 08/06/2021

A class template that describes an object that can serve as a locale facet. It can control conversions between a sequence of values used to encode characters within the program and a sequence of values used to encode characters outside the program.

Syntax

C++

```
template <class CharType, class Byte, class StateType>
class codecvt : public locale::facet, codecvt_base;
```

Parameters

`CharType`

The type used within a program to encode characters.

`Byte`

A type used to encode characters outside a program.

`StateType`

A type that can be used to represent intermediate states of a conversion between internal and external types of character representations.

Remarks

The class template describes an object that can serve as a locale facet, to control conversions between a sequence of values of type *`CharType`* and a sequence of values of type *`Byte`*. The class *`StateType`* characterizes the transformation -- and an object of class *`StateType`* stores any necessary state information during a conversion.

The internal encoding uses a representation with a fixed number of bytes per character, usually either type **`char`** or type **`wchar_t`**.

As with any locale facet, the static object `id` has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The template versions of do_in and do_out always return `codecvt_base::noconv`.

The C++ Standard Library defines several explicit specializations:

```
C++

template<>
codecvt<wchar_t, char, mbstate_t>
```

converts between `wchar_t` and `char` sequences.

```
C++

template<>
codecvt<char16_t, char, mbstate_t>
```

converts between `char16_t` sequences encoded as UTF-16 and `char` sequences encoded as UTF-8.

```
C++

template<>
codecvt<char32_t, char, mbstate_t>
```

converts between `char32_t` sequences encoded as UTF-32 (UCS-4) and `char` sequences encoded as UTF-8.

Constructors

Constructor	Description
codecvt	The constructor for objects of class `codecvt` that serves as a locale facet to handle conversions.

Typedefs

Type name	Description
extern_type	A character type that is used for external representations.
intern_type	A character type that is used for internal representations.
state_type	A character type that is used to represent intermediate states during conversions between internal and external representations.

Member functions

Member function	Description
always_noconv	Tests whether no conversions need be done.
do_always_noconv	A virtual function called to test whether no conversions need be done.
do_encoding	A virtual function that tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte` values used and the `CharType` values produced is constant, and, if so, determines the value of that ratio.
do_in	A virtual function called to convert a sequence of internal `Byte` values to a sequence of external `CharType` values.
do_length	A virtual function that determines how many `Byte` values from a given sequence of external `Byte` values produce not more than a given number of internal `CharType` values and returns that number of `Byte` values.
do_max_length	A virtual function that returns the maximum number of external Bytes necessary to produce one internal `CharType`.
do_out	A virtual function called to convert a sequence of internal `CharType` values to a sequence of external Bytes.
do_unshift	A virtual function called to provide the `Byte` values needed in a state-dependent conversion to complete the last character in a sequence of `Byte` values.
encoding	Tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte` values used and the `CharType` values produced is constant, and, if so, determines the value of that ratio.
in	Converts an external representation of a sequence of `Byte` values to an internal representation of a sequence of `CharType` values.
length	Determines how many `Byte` values from a given sequence of external `Byte` values produce not more than a given number of internal `CharType` values and returns that number of `Byte` values.
max_length	Returns the maximum number of external `Byte` values necessary to produce one internal `CharType`.
out	Converts a sequence of internal `CharType` values to a sequence of external `Byte` values.
unshift	Provides the external `Byte` values needed in a state-dependent conversion to complete the last character in the sequence of `Byte` values.

Requirements

Header: `<locale>`

Namespace: `std`

`codecvt::always_noconv`

Tests whether no conversions need to be done.

C++

```
bool always_noconv() const throw();
```

Return Value

A Boolean value that is **`true`** if no conversions need to be done; **`false`** if at least one needs to be done.

Remarks

The member function returns do_always_noconv.

Example

C++

```
// codecvt_always_noconv.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   bool result1 = use_facet<codecvt<char, char, mbstate_t>>
      ( loc ).always_noconv( );

   if ( result1 )
      cout << "No conversion is needed." << '\n';
   else
      cout << "At least one conversion is required." << '\n';

   bool result2 = use_facet<codecvt<wchar_t, char, mbstate_t>>
```

```
      ( loc ).always_noconv( );

   if ( result2 )
      cout << "No conversion is needed." << '\n';
   else
      cout << "At least one conversion is required." << '\n';
}
```

Output

```
No conversion is needed.
At least one conversion is required.
```

codecvt::codecvt

The constructor for objects of class codecvt that serves as a locale facet to handle conversions.

C++

```
explicit codecvt(size_t refs = 0);
```

Parameters

refs

Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- 2: These values aren't defined.

The constructor initializes its `locale::facet` base object with locale::facet`(refs)`.

codecvt::do_always_noconv

A virtual function called to test whether no conversions need to be done.

C++

```
virtual bool do_always_noconv() const throw();
```

Return Value

The protected virtual member function returns `true` only if every call to do_in or do_out returns `noconv`.

The template version always returns `true`.

Example

See the example for always_noconv, which calls `do_always_noconv`.

`codecvt::do_encoding`

A virtual function that tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte` values used and the `CharType` values produced is constant and, if so, determines the value of that ratio.

C++

```
virtual int do_encoding() const throw();
```

Return Value

The protected virtual member function returns:

- -1, if the encoding of sequences of type `extern_type` is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- `N`, if the encoding involves only sequences of length `N`

Example

See the example for encoding, which calls `do_encoding`.

codecvt::do_in

A virtual function called to convert a sequence of external `Byte` values to a sequence of internal `CharType` values.

```
C++
```

```
virtual result do_in(
    StateType& state,
    const Byte* first1,
    const Byte* last1,
    const Byte*& next1,
    CharType* first2,
    CharType* last2,
    CharType*& next2,) const;
```

Parameters

`state`

The conversion state that is maintained between calls to the member function.

`first1`

Pointer to the beginning of the sequence to be converted.

`last1`

Pointer to the end of the sequence to be converted.

`next1`

Pointer beyond the end of the converted sequence, to the first unconverted character.

`first2`

Pointer to the beginning of the converted sequence.

`last2`

Pointer to the end of the converted sequence.

`next2`

Pointer to the `CharType` that comes after the last converted `CharType`, to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success, or failure of the operation. The function returns:

- `codecvt_base::error` if the source sequence is ill formed.

- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination isn't large enough, for the conversion to succeed.

Remarks

`state` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for in, which calls `do_in`.

`codecvt::do_length`

A virtual function that determines how many `Byte` values from a given sequence of external `Byte` values produce not more than a given number of internal `CharType` values and returns that number of `Byte` values.

```
C++
```

```
virtual int do_length(
    const StateType& state,
    const Byte* first1,
    const Byte* last1,
    size_t len2) const;
```

Parameters

`state`

The conversion state that is maintained between calls to the member function.

`first1`

Pointer to the beginning of the external sequence.

`last1`

Pointer to the end of the external sequence.

len2

The maximum number of `Byte` values that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *len2*, defined by the external source sequence at [`first1`, `last1`).

Remarks

The protected virtual member function effectively calls `do_in( state, first1, last1, next1, buf, buf + len2, next2)` for *state* (a copy of state), some buffer `buf`, and pointers `next1` and `next2`.

It then returns `next2` - `buf`. It counts the maximum number of conversions, not greater than *len2*, defined by the source sequence at [`first1`, `last1`).

The template version always returns the lesser of *last1* - *first1* and *len2*.

Example

See the example for length, which calls `do_length`.

codecvt::do_max_length

A virtual function that returns the maximum number of external `Byte` values necessary to produce one internal `CharType`.

```
C++
```

```
virtual int do_max_length() const throw();
```

Return Value

The maximum number of `Byte` values necessary to produce one `CharType`.

Remarks

The protected virtual member function returns the largest permissible value that can be returned by do_length`( first1, last1, 1)` for arbitrary valid values of *`first1`* and *`last1`*.

Example

See the example for max_length, which calls `do_max_length`.

`codecvt::do_out`

A virtual function called to convert a sequence of internal `CharType` values to a sequence of external `Byte` values.

```
C++
```

```
virtual result do_out(
    StateType& state,
    const CharType* first1,
    const CharType* last1,
    const CharType*& next1,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

`state`

The conversion state that is maintained between calls to the member function.

`first1`

Pointer to the beginning of the sequence to be converted.

`last1`

Pointer to the end of the sequence to be converted.

`next1`

Reference to a pointer to the first unconverted `CharType`, after the last `CharType` converted.

`first2`

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Reference to a pointer to the first unconverted `Byte`, after the last `Byte` converted.

Return Value

The function returns:

- `codecvt_base::error` if the source sequence is ill formed.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination isn't large enough for the conversion to succeed.

Remarks

state must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

Example

See the example for out, which calls `do_out`.

`codecvt::do_unshift`

A virtual function called to provide the `Byte` values needed in a state-dependent conversion to complete the last character in a sequence of `Byte` values.

C++

```
virtual result do_unshift(
    StateType& state,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

state

The conversion state that is maintained between calls to the member function.

first2

Pointer to the first position in the destination range.

last2

Pointer to the last position in the destination range.

next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- `codecvt_base::error` if *state* represents an invalid state
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the destination isn't large enough for the conversion to succeed

Remarks

The protected virtual member function tries to convert the source element `CharType`(0) to a destination sequence that it stores within [`first2`, `last2`), except for the terminating element `Byte`(0). It always stores in *`next2`* a pointer to the first unaltered element in the destination sequence.

`State` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value as needed to reflect the current state of a successful conversion. Typically, converting the source element `CharType`(0) leaves the current state in the initial conversion state.

Example

See the example for unshift, which calls `do_unshift`.

`codecvt::encoding`

Tests if the encoding of the `Byte` stream is state dependent, whether the ratio between the `Byte` values used and the `CharType` values produced is constant, and, if so, determines the value of that ratio.

```
C++
```

```
int encoding() const throw();
```

Return Value

If the return value is positive, that value is the constant number of `Byte` characters required to produce the `CharType` character.

The protected virtual member function returns:

- -1, if the encoding of sequences of type `extern_type` is state dependent.
- 0, if the encoding involves sequences of varying lengths.
- `N`, if the encoding involves only sequences of length `N`.

Remarks

The member function returns do_encoding.

Example

```
C++
```

```
// codecvt_encoding.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   int result1 = use_facet<codecvt<char, char, mbstate_t>> ( loc ).encoding
( );
   cout << result1 << '\n';
   result1 = use_facet<codecvt<wchar_t, char, mbstate_t>> ( loc ).encoding(
);
```

```
    cout << result1 << '\n';
    result1 = use_facet<codecvt<char, wchar_t, mbstate_t>> ( loc ).encoding(
);
    cout << result1 << '\n';
}
```

Output

```
1
1
1
```

`codecvt::extern_type`

A character type that is used for external representations.

C++

```
typedef Byte extern_type;
```

Remarks

The type is a synonym for the template parameter `Byte`.

codecvt::in

Converts an external representation of a sequence of `Byte` values to an internal representation of a sequence of `CharType` values.

C++

```
result in(
    StateType& state,
    const Byte* first1,
    const Byte* last1,
    const Byte*& next1,
    CharType* first2,
    CharType* last2,
    CharType*& next2,) const;
```

Parameters

state
The conversion state that is maintained between calls to the member function.

first1
Pointer to the beginning of the sequence to be converted.

last1
Pointer to the end of the sequence to be converted.

next1
Pointer beyond the end of the converted sequence to the first unconverted character.

first2
Pointer to the beginning of the converted sequence.

last2
Pointer to the end of the converted sequence.

next2
Pointer to the `CharType` that comes after the last converted `Chartype` to the first unaltered character in the destination sequence.

Return Value

A return that indicates the success, partial success, or failure of the operation. The function returns:

- `codecvt_base::error` if the source sequence is ill formed.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the source is insufficient or if the destination isn't large enough for the conversion to succeed.

Remarks

state must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. After a partial conversion, *state* must be set so as to allow the conversion to resume when new characters arrive.

The member function returns do_in(state, first1, last1, next1, first2, last2, next2).

Example

C++

```
// codecvt_in.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
   const char* pszExt = "This is the string to be converted!";
   wchar_t pwszInt [LEN+1];
   memset(&pwszInt[0], 0, (sizeof(wchar_t))*(LEN+1));
   const char* pszNext;
   wchar_t* pwszNext;
   mbstate_t state = {0}; // zero-initialization represents the initial
conversion state for mbstate_t
   locale loc("C");//English_Britain");//German_Germany
   int res = use_facet<codecvt<wchar_t, char, mbstate_t>>
     ( loc ).in( state,
          pszExt, &pszExt[strlen(pszExt)], pszNext,
          pwszInt, &pwszInt[strlen(pszExt)], pwszNext );
   pwszInt[strlen(pszExt)] = 0;
   wcout << ( res!=codecvt_base::error ?  L"It worked! " : L"It didn't work!
" )
       << L"The converted string is:\n ["
       << &pwszInt[0]
       << L"]" << '\n';
   exit(-1);
}
```

Output

```
It worked! The converted string is:
[This is the string to be converted!]
```

codecvt::intern_type

A character type that is used for internal representations.

C++

```
typedef CharType intern_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

codecvt::length

Determines how many `Byte` values from a given sequence of external `Byte` values produce not more than a given number of internal `CharType` values and returns that number of `Byte` values.

```
C++
```

```
int length(
    const StateType& state,
    const Byte* first1,
    const Byte* last1,
    size_t len2) const;
```

Parameters

`state`
The conversion state that is maintained between calls to the member function.

`first1`
Pointer to the beginning of the external sequence.

`last1`
Pointer to the end of the external sequence.

`len2`
The maximum number of Bytes that can be returned by the member function.

Return Value

An integer that represents a count of the maximum number of conversions, not greater than *`len2`*, defined by the external source sequence at [`first1`, `last1`).

Remarks

The member function returns do_length(state, first1, last1, len2).

Example

C++

```
// codecvt_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;
#define LEN 90
int main( )
{
   const char* pszExt = "This is the string whose length is to be
measured!";
   mbstate_t state = {0}; // zero-initialization represents the initial
conversion state for mbstate_t
   locale loc("C"); // English_Britain"); //German_Germany
   int res = use_facet<codecvt<wchar_t, char, mbstate_t>>
     ( loc ).length( state,
          pszExt, &pszExt[strlen(pszExt)], LEN );
   cout << "The length of the string is: ";
   wcout << res;
   cout << "." << '\n';
   exit(-1);
}
```

Output

```
The length of the string is: 50.
```

codecvt::max_length

Returns the maximum number of external `Byte` values necessary to produce one internal `CharType`.

C++

```
int max_length() const throw();
```

Return Value

The maximum number of `Byte` values necessary to produce one `CharType`.

Remarks

The member function returns do_max_length.

Example

C++

```
// codecvt_max_length.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc( "C");//English_Britain" );//German_Germany
   int res = use_facet<codecvt<char, char, mbstate_t>>
     ( loc ).max_length( );
   wcout << res << '\n';
}
```

Output

```
1
```

codecvt::out

Converts a sequence of internal `CharType` values to a sequence of external `Byte` values.

C++

```
result out(
    StateType& state,
    const CharType* first1,
    const CharType* last1,
    const CharType*& next1,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

state

The conversion state that is maintained between calls to the member function.

first1

Pointer to the beginning of the sequence to be converted.

last1

Pointer to the end of the sequence to be converted.

next1

Reference to a pointer to the first unconverted `CharType` after the last `CharType` converted.

first2

Pointer to the beginning of the converted sequence.

last2

Pointer to the end of the converted sequence.

next2

Reference to a pointer to the first unconverted `Byte` after the last converted `Byte`.

Return Value

The member function returns do_out`( state, first1, last1, next1, first2, last2, next2)`.

Remarks

For more information, see codecvt::do_out.

Example

C++

```
// codecvt_out.cpp
// compile with: /EHsc
#define _INTL
#include <locale>
#include <iostream>
#include <wchar.h>
using namespace std;
#define LEN 90
int main( )
```

```
{
    char pszExt[LEN + 1];
    const wchar_t* pwszInt = L"This is the wchar_t string to be converted.";
    memset(&pszExt[0], 0, (sizeof(char)) * (LEN + 1));
    char* pszNext;
    const wchar_t* pwszNext;
    mbstate_t state;
    locale loc("C");//English_Britain");//German_Germany
    int res = use_facet<codecvt<wchar_t, char, mbstate_t>>
        (loc).out(state,
            pwszInt, &pwszInt[wcslen(pwszInt)], pwszNext,
            pszExt, &pszExt[wcslen(pwszInt)], pszNext);
    pszExt[wcslen(pwszInt)] = 0;
    cout << (res != codecvt_base::error ? "It worked: " : "It didn't work:
")
        << "The converted string is:\n ["
        << &pszExt[0]
        << "]" << '\n';

}
```

Output

```
It worked: The converted string is:
[This is the wchar_t string to be converted.]
```

`codecvt::state_type`

A character type that is used to represent intermediate states during conversions between internal and external representations.

C++

```
typedef StateType state_type;
```

Remarks

The type is a synonym for the template parameter `StateType`.

codecvt::unshift

Provides the `Byte` values needed in a state-dependent conversion to complete the last character in a sequence of `Byte` values.

```
C++
```

```
result unshift(
    StateType& state,
    Byte* first2,
    Byte* last2,
    Byte*& next2) const;
```

Parameters

state

The conversion state that is maintained between calls to the member function.

first2

Pointer to the first position in the destination range.

last2

Pointer to the last position in the destination range.

next2

Pointer to the first unaltered element in the destination sequence.

Return Value

The function returns:

- `codecvt_base::error` if state represents an invalid state.
- `codecvt_base::noconv` if the function performs no conversion.
- `codecvt_base::ok` if the conversion succeeds.
- `codecvt_base::partial` if the destination isn't large enough for the conversion to succeed.

Remarks

The protected virtual member function tries to convert the source element `CharType`(0) to a destination sequence that it stores within [`first2`, `last2`), except for the terminating element `Byte`(0). It always stores in *next2* a pointer to the first unaltered element in the destination sequence.

state must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Typically, converting the source element CharType(0) leaves the current state in the initial conversion state.

The member function returns do_unshift(state, first2, last2, next2).

See also

<locale>
Code Pages
Locale Names, Languages, and Country/Region Strings
Thread Safety in the C++ Standard Library

codecvt_base Class

Article • 08/03/2021

A base class for the codecvt class that is used to define an enumeration type referred to as `result`, used as the return type for the facet member functions to indicate the result of a conversion.

Syntax

C++

```
class codecvt_base : public locale::facet {
public:
    enum result {ok, partial, error, noconv};
    codecvt_base( size_t _Refs = 0);
    bool always_noconv() const;
    int max_length() const;
    int encoding() const;
    ~codecvt_base()

protected:
    virtual bool do_always_noconv() const;
    virtual int do_max_length() const;
    virtual int do_encoding() const;
};
```

Remarks

The class describes an enumeration common to all specializations of class template codecvt. The enumeration result describes the possible return values from do_in or do_out:

- `ok` if the conversion between internal and external character encodings succeeds.
- `partial` if the destination is not large enough for the conversion to succeed.
- `error` if the source sequence is ill formed.
- `noconv` if the function performs no conversion.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

`codecvt_byname` class

Article • 06/30/2022

A derived class template that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning conversions.

Syntax

C++

```
template <class CharType, class Byte, class StateType>
class codecvt_byname: public codecvt<CharType, Byte, StateType> {
public:
    explicit codecvt_byname(
    const char* locale_name,
    size_t ref_count = 0);

explicit codecvt_byname(
    const string& locale_name,
    size_t ref_count = 0);

protected:
    virtual ~codecvt_byname();

};
```

Parameters

`locale_name`

A named locale.

`ref_count`

An initial reference count.

Remarks

`byname` facets are automatically created when a named locale is constructed.

Its behavior is determined by the named locale *`locale_name`*. Each constructor initializes its base object by using `codecvt<CharType, Byte, StateType>( ref_count )`. For more information, see codecvt.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

collate Class

Article • 08/03/2021

A class template that describes an object that can serve as a locale facet to control the ordering and grouping of characters within a string, comparisons between them and the hashing of strings.

Syntax

C++

```
template <class CharType>
class collate : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`. In some languages, characters are grouped and treated as a single character, and in others, individual characters are treated as if they were two characters. The collating services provided by the collate class provide the way to sort these cases.

Constructors

Constructor	Description
collate	The constructor for objects of class `collate` that serves as a locale facet to handle string sorting conventions.

Typedefs

Type name	Description

Type name	Description
char_type	A type that describes a character of type `CharType`.
string_type	A type that describes a string of type `basic_string` containing characters of type `CharType`.

Member functions

Member function	Description
compare	Compares two character sequences according to their facet-specific rules for equality or inequality.
do_compare	A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.
do_hash	A virtual function called to determine the hash value of sequences according to their facet-specific rules.
do_transform	A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.
hash	Determines the hash value of sequence according to their facet-specific rules.
transform	Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

Requirements

Header: <locale>

Namespace: std

collate::char_type

A type that describes a character of type `CharType`.

```
C++
```

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

collate::collate

The constructor for objects of class collate that serves as a locale facet to handle string sorting conventions.

```
C++
```

```
public:
    explicit collate(
    size_t _Refs = 0);

protected:
    collate(
const char* _Locname,
    size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

_Locname
The name of the locale.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

The constructor initializes its base object with **locale::**facet(`_Refs`).

collate::compare

Compares two character sequences according to their facet-specific rules for equality or inequality.

```
C++
```

```
int compare(const CharType* first1,
    const CharType* last1,
    const CharType* first2,
    const CharType* last2) const;
```

Parameters

first1
Pointer to the first element in the first sequence to be compared.

last1
Pointer to the last element in the first sequence to be compared.

first2
Pointer to the first element in the second sequence to be compared.

last2
Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or, if no unequal pairs exist, but the first sequence is shorter.

The member function returns do_compare(`first1`, `last1`, `first2`, `last2`).

Example

C++

```
// collate_compare.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-
s, it comes before z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare ( s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << result1 << endl;

   locale loc2 ( "C" );
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << result2 << endl;
}
```

collate::do_compare

A virtual function called to compare two character sequences according to their facet-specific rules for equality or inequality.

C++

```
virtual int do_compare(const CharType* first1,
    const CharType* last1,
    const CharType* first2,
    const CharType* last2) const;
```

Parameters

first1
Pointer to the first element in the first sequence to be compared.

last1
Pointer to the last element in the first sequence to be compared.

first2
Pointer to the first element in the second sequence to be compared.

last2
Pointer to the last element in the second sequence to be compared.

Return Value

The member function returns:

- -1 if the first sequence compares less than the second sequence.
- +1 if the second sequence compares less than the first sequence.
- 0 if the sequences are equivalent.

Remarks

The protected virtual member function compares the sequence at [* first1, Last1)* with the sequence at *[first2, last2*). It compares values by applying `operator<` between pairs of corresponding elements of type `CharType`. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences or if no unequal pairs exist but the first sequence is shorter.

Example

See the example for collate::compare, which calls `do_compare`.

collate::do_hash

A virtual function called to determine the hash value of sequences according to their facet-specific rules.

```
C++

virtual long do_hash(const CharType* first, const CharType* last) const;
```

Parameters

first
A pointer to the first character in the sequence whose has value is to be determined.

last
A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type `long` for the sequence.

Remarks

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

See the example for hash, which calls `do_hash`.

collate::do_transform

A virtual function called to convert a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
C++
```

```
virtual string_type do_transform(const CharType* first, const CharType*
last) const;
```

Parameters

first
A pointer to the first character in the sequence to be converted.

last
A pointer to the last character in the sequence to be converted.

Return Value

A string that is the transformed character sequence.

Remarks

The protected virtual member function returns an object of class string_type whose controlled sequence is a copy of the sequence [`first`, `last`). If a class derived from

collate< **CharType**> overrides do_compare, it should also override `do_transform` to match. When passed to `collate::compare`, two transformed strings should yield the same result that you would get from passing the untransformed strings to compare in the derived class.

Example

See the example for transform, which calls `do_transform`.

collate::hash

Determines the hash value of sequence according to their facet-specific rules.

```
C++
```

```
long hash(const CharType* first, const CharType* last) const;
```

Parameters

first
A pointer to the first character in the sequence whose has value is to be determined.

last
A pointer to the last character in the sequence whose has value is to be determined.

Return Value

A hash value of type `long` for the sequence.

Remarks

The member function returns do_hash(`first`, `last`).

A hash value can be useful, for example, in distributing sequences pseudo-randomly across an array of lists.

Example

```
C++
```

```
// collate_hash.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("\x00dfzz abc."); // \x00df is the German sharp-s (looks
like beta), it comes before z in the alphabet
   _TCHAR * s2 = _T("zzz abc."); // \x00df is the German sharp-s (looks like
beta), it comes before z in the alphabet

   long r1 = use_facet< collate<_TCHAR> > ( loc ).
      hash (s1, &s1[_tcslen( s1 )-1 ]);
   long r2 =  use_facet< collate<_TCHAR> > ( loc ).
      hash (s2, &s2[_tcslen( s2 )-1 ] );
   cout << r1 << " " << r2 << endl;
}
```

Output

```
541187293 551279837
```

collate::string_type

A type that describes a string of type `basic_string` containing characters of type `CharType`.

C++

```
typedef basic_string<CharType> string_type;
```

Remarks

The type describes a specialization of class template basic_string whose objects can store copies of the source sequence.

Example

For an example of how to declare and use `string_type`, see transform.

collate::transform

Converts a character sequence from a locale to a string that may be used in lexicographical comparisons with other character sequences similarly converted from the same locale.

```
C++
```

```
string_type transform(const CharType* first, const CharType* last) const;
```

Parameters

first
A pointer to the first character in the sequence to be converted.

last
A pointer to the last character in the sequence to be converted.

Return Value

A string that contains the transformed character sequence.

Remarks

The member function returns do_transform(`first`, `last`).

Example

```
C++
```

```
// collate_transform.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( )
{
   locale loc ( "German_Germany" );
   _TCHAR* s1 = _T("\x00dfzz abc.");
   // \x00df is the German sharp-s (looks like beta),
   // it comes before z in the alphabet
   _TCHAR* s2 = _T("zzz abc.");
```

```
   collate<_TCHAR>::string_type r1;   // OK for typedef
   r1 = use_facet< collate<_TCHAR> > ( loc ).
      transform (s1, &s1[_tcslen( s1 )-1 ]);

   cout << r1 << endl;

   basic_string<_TCHAR> r2 = use_facet< collate<_TCHAR> > ( loc ).
      transform (s2, &s2[_tcslen( s2 )-1 ]);

   cout << r2 << endl;

   int result1 = use_facet<collate<_TCHAR> > ( loc ).compare
      (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );

   cout << _tcscmp(r1.c_str( ),r2.c_str( )) << result1
      << _tcscmp(s1,s2) <<endl;
}
```

Output

```
-1-11
```

See also

<locale>

Thread Safety in the C++ Standard Library

collate_byname Class

Article • 08/03/2021

A derived class template that describes an object that can serve as a collate facet of a given locale, enabling the retrieval of information specific to a cultural area concerning string sorting conventions.

Syntax

C++

```
template <class CharType>
class collate_byname : public collate<CharType> {
public:
    explicit collate_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit collate_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~collate_byname();

};
```

Parameters

_Locname
A named locale.

_Refs
An initial reference count.

Remarks

The class template describes an object that can serve as a locale facet of type collate<CharType>. Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with collate<CharType>(`_Refs`).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

ctype Class

Article • 05/20/2022

A class that provides a facet that is used to classify characters, convert from upper and lower cases, and convert between the native character set and that set used by the locale.

Syntax

C++

```
template <class CharType>
class ctype : public ctype_base;
```

Parameters

CharType
The type used within a program to encode characters.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`. Classification criteria are provided a nested bitmask type in the base class ctype_base.

The C++ Standard Library defines two explicit specializations of this class template:

- `ctype<char>`, an explicit specialization whose differences are described separately. For more information, see ctype<char> Class.
- `ctype<wchar_t>`, which treats elements as wide characters.

Other specializations of class template `ctype<CharType>`:

- Convert a value *ch* of type *CharType* to a value of type `char` with the expression `(char)ch`.
- Convert a value *byte* of type `char` to a value of type *CharType* with the expression `CharType(byte)`.

All other operations are performed on `char` values in the same way as for the explicit specialization `ctype<char>`.

Constructors

Constructor	Description
ctype	Constructor for objects of class `ctype` that serve as locale facets for characters.

Typedefs

Type name	Description
char_type	A type that describes a character used by a locale.

Member functions

Member function	Description
do_is	A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.
do_narrow	A virtual function called to convert a character of type `CharType` used by a locale to the corresponding character of type `char` in the native character set.
do_scan_is	A virtual function called to locate the first character in a range that matches a specified mask.
do_scan_not	A virtual function called to locate the first character in a range that does not match a specified mask.
do_tolower	A virtual function called to convert a character or a range of characters to their lower case.
do_toupper	A virtual function called to convert a character or a range of characters to upper case.
do_widen	A virtual function called to converts a character of type `char` in the native character set to the corresponding character of type `CharType` used by a locale.
is	Tests whether a single character has a particular attribute, or classifies the attributes of each character in a range and stores them in an array.
narrow	Converts a character of type `CharType` used by a locale to the corresponding character of type char in the native character set.

Member function	Description
scan_is	Locates the first character in a range that matches a specified mask.
scan_not	Locates the first character in a range that does not match a specified mask.
tolower	Converts a character or a range of characters to lower case.
toupper	Converts a character or a range of characters to upper case.
widen	Converts a character of type `char` in the native character set to the corresponding character of type `CharType` used by a locale.

Requirements

Header: <locale>

Namespace: std

ctype::char_type

A type that describes a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

Example

See the member function widen for an example that uses `char_type` as a return value.

ctype::ctype

Constructor for objects of class ctype that serve as locale facets for characters.

```
C++

explicit ctype(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its `locale::facet` base object with **locale::**facet(`_Refs`).

ctype::do_is

A virtual function called to test whether a single character has a particular attribute, or classify the attributes of each character in a range and stores them in an array.

C++

```
virtual bool do_is(
    mask maskVal,
    CharType ch) const;

virtual const CharType *do_is(
    const CharType* first,
    const CharType* last,
    mask* dest) const;
```

Parameters

maskVal
The mask value for which the character is to be tested.

ch
The character whose attributes are to be tested.

first
A pointer to the first character in the range whose attributes are to be classified.

last
A pointer to the character immediately following the last character in the range whose attributes are to be classified.

dest
A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns a Boolean value that is `true` if the character tested has the attribute described by the mask value; `false` if it fails to have the attribute.

The second member function returns an array containing the mask values characterizing the attributes of each of the characters in the range.

Remarks

The mask values classifying the attributes of the characters are provided by the class ctype_base, from which ctype derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the bitwise operators (`|`, `&`, `^`, `~`).

Example

See the example for is, which calls `do_is`.

ctype::do_narrow

A virtual function called to convert a character of type `CharType` used by a locale to the corresponding character of type `char` in the native character set.

C++

```
virtual char do_narrow(
    CharType ch,
    char default = '\0') const;

virtual const CharType* do_narrow(
    const CharType* first,
```

```
    const CharType* last,
    char default,
    char* dest) const;
```

Parameters

ch
The character of type `Chartype` used by the locale to be converted.

default
The default value to be assigned by the member function to characters of type `CharType` that do not have counterpart characters of type **`char`**.

first
A pointer to the first character in the range of characters to be converted.

last
A pointer to the character immediately following the last character in the range of characters to be converted.

dest
A const pointer to the first character of type **`char`** in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the native character of type char that corresponds to the parameter character of type `CharType` or *default* if no counterpart is defined.

The second protected member function returns a pointer to the destination range of native characters converted from characters of type `CharType`.

Remarks

The second protected member template function stores in `dest`[`I`] the value `do_narrow`(`first` [`I`], `default`), for `I` in the interval [0, `last` - `first`).

Example

See the example for narrow, which calls `do_narrow`.

ctype::do_scan_is

A virtual function called to locate the first character in a range that matches a specified mask.

```
C++
```

```
virtual const CharType *do_scan_is(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal
The mask value to be matched by a character.

first
A pointer to the first character in the range to be scanned.

last
A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *last*.

Remarks

The protected member function returns the smallest pointer `ptr` in the range [`first`, `last`) for which do_is(`maskVal`, * `ptr`) is true.

Example

See the example for scan_is, which calls `do_scan_is`.

ctype::do_scan_not

A virtual function called to locate the first character in a range that does not match a specified mask.

```
C++
```

```
virtual const CharType *do_scan_not(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal
The mask value not to be matched by a character.

first
A pointer to the first character in the range to be scanned.

last
A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that doesn't match a specified mask. If no such value exists, the function returns *last*.

Remarks

The protected member function returns the smallest pointer `ptr` in the range [`first`, `last`) for which do_is(`maskVal`, * `ptr`) is false.

Example

See the example for scan_not, which calls `do_scan_not`.

ctype::do_tolower

A virtual function called to convert a character or a range of characters to lower case.

```
C++
```

```
virtual CharType do_tolower(CharType ch) const;

virtual const CharType *do_tolower(
    CharType* first,
    const CharType* last) const;
```

Parameters

ch
The character to be converted to lower case.

first
A pointer to the first character in the range of characters whose cases are to be converted.

last
A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the lowercase form of the parameter *ch*. If no lowercase form exists, it returns *ch*. The second protected member function returns *last*.

Remarks

The second protected member template function replaces each element `first` [`I`], for `I` in the interval [0, `last` - `first`), with `do_tolower`(`first` [`I`]).

Example

See the example for tolower, which calls `do_tolower`.

ctype::do_toupper

A virtual function called to convert a character or a range of characters to upper case.

```
C++
```

```
virtual CharType do_toupper(CharType ch) const;

virtual const CharType *do_toupper(
    CharType* first,
    const CharType* last) const;
```

Parameters

ch
The character to be converted to upper case.

first
A pointer to the first character in the range of characters whose cases are to be converted.

last
A pointer to character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first protected member function returns the uppercase form of the parameter *ch*. If no uppercase form exists, it returns *ch*. The second protected member function returns *last*.

Remarks

The second protected member template function replaces each element `first` [`I`], for `I` in the interval [0, `last` - `first`), with `do_toupper`(`first` [`I`]).

Example

See the example for toupper, which calls `do_toupper`.

ctype::do_widen

A virtual function called to converts a character of type **`char`** in the native character set to the corresponding character of type `CharType` used by a locale.

```
C++
```

```
virtual CharType do_widen(char byte) const;

virtual const char *do_widen(
    const char* first,
    const char* last,
    CharType* dest) const;
```

Parameters

byte
The character of type **`char`** in the native character set to be converted.

first
A pointer to the first character in the range of characters to be converted.

last
A pointer to the character immediately following the last character in the range of characters to be converted.

dest
A pointer to the first character of type `CharType` in the destination range that stores the converted range of characters.

Return Value

The first protected member function returns the character of type `CharType` that corresponds to the parameter character of native type **`char`**.

The second protected member function returns a pointer to the destination range of characters of type `CharType` used by a locale converted from native characters of type **`char`**.

Remarks

The second protected member template function stores in `dest`[`I`] the value `do_widen`(`first`[`I`]), for `I` in the interval [0, `last` - `first`).

Example

See the example for widen, which calls `do_widen`.

ctype::is

Tests whether a single character has a particular attribute or classifies the attributes of each character in a range and stores them in an array.

```
C++
```

```
bool is(mask maskVal, CharType ch) const;

const CharType *is(
    const CharType* first,
    const CharType* last,
    mask* dest) const;
```

Parameters

maskVal
The mask value for which the character is to be tested.

ch
The character whose attributes are to be tested.

first
A pointer to the first character in the range whose attributes are to be classified.

last
A pointer to the character immediately following the last character in the range whose attributes are to be classified.

dest
A pointer to the beginning of the array where the mask values characterizing the attributes of each of the characters are to be stored.

Return Value

The first member function returns `true` if the character tested has the attribute described by the mask value; `false` if it fails to have the attribute.

The second member function returns a pointer to the last character in the range whose attributes are to be classified.

Remarks

The mask values classifying the attributes of the characters are provided by the class ctype_base Class, from which ctype derives. The first member function can accept expressions for its first parameter referred to as bitmasks and formed from the combination of mask values by the bitwise operators (`|`, `&`, `^`, `~`).

Example

C++

```
// ctype_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main() {
   locale loc1 ( "German_Germany" ), loc2 ( "English_Australia" );

   if (use_facet<ctype<char> > ( loc1 ).is( ctype_base::alpha, 'a' ))
      cout << "The character 'a' in locale loc1 is alphabetic."
           << endl;
   else
      cout << "The character 'a' in locale loc1 is not alphabetic."
           << endl;

   if (use_facet<ctype<char> > ( loc2 ).is( ctype_base::alpha, '!' ))
      cout << "The character '!' in locale loc2 is alphabetic."
           << endl;
   else
      cout << "The character '!' in locale loc2 is not alphabetic."
           << endl;

   char *string = "Hello, my name is John!";
   ctype<char>::mask maskarray[30];
   use_facet<ctype<char> > ( loc2 ).is(
      string, string + strlen(string), maskarray );
   for (unsigned int i = 0; i < strlen(string); i++) {
      cout << string[i] << ": "
           << (maskarray[i] & ctype_base::alpha  "alpha"
                                                 : "not alpha")
           << endl;;
   };
}
```

ctype::narrow

Converts characters of type `CharType` used by a locale to the corresponding characters of type **`char`** in the native character set.

C++

```
char narrow(CharType ch, char default = '\0') const;

const CharType* narrow(
    const CharType* first,
    const CharType* last,
    char default,
    char* dest) const;
```

Parameters

ch
The character of type `Chartype` used by the locale to be converted.

default
The default value to be assigned by the member function to characters of type `CharType` that do not have counterpart characters of type **`char`**.

first
A pointer to the first character in the range of characters to be converted.

last
A pointer to the character immediately following the last character in the range of characters to be converted.

dest
A const pointer to the first character of type **`char`** in the destination range that stores the converted range of characters.

Return Value

The first member function returns the native character of type **`char`** that corresponds to the parameter character of type `CharType default` if not counterpart is defined.

The second member function returns a pointer to the destination range of native characters converted from characters of type `CharType`.

Remarks

The first member function returns do_narrow(`ch`, `default`). The second member function returns do_narrow (`first`, `last`, `default`, `dest`). Only the basic source characters are guaranteed to have a unique inverse image `CharType` under `narrow`. For

these basic source characters, the following invariant holds: `narrow` (widen (**c**), 0) == **c**.

Example

C++

```
// ctype_narrow.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "english" );
   wchar_t *str1 = L"\x0392fhello everyone";
   char str2 [16];
   bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).narrow
      ( str1, str1 + wcslen(str1), 'X', &str2[0] ) != 0);  // C4996
   str2[wcslen(str1)] = '\0';
   wcout << str1 << endl;
   cout << &str2[0] << endl;
}
```

Output

```
Xhello everyone
```

ctype::scan_is

Locates the first character in a range that matches a specified mask.

C++

```
const CharType *scan_is(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal
The mask value to be matched by a character.

first
A pointer to the first character in the range to be scanned.

last
A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does match a specified mask. If no such value exists, the function returns *last*.

Remarks

The member function returns do_scan_is(`maskVal`, `first`, `last`).

Example

C++

```
// ctype_scan_is.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char *string = "Hello, my name is John!";

   const char* i = use_facet<ctype<char> > ( loc1 ).scan_is
      ( ctype_base::punct, string, string + strlen(string) );
   cout << "The first punctuation is \"" << *i << "\" at position: "
      << i - string << endl;
}
```

Output

```
The first punctuation is "," at position: 5
```

ctype::scan_not

Locates the first character in a range that does not match a specified mask.

```
C++
```

```
const CharType *scan_not(
    mask maskVal,
    const CharType* first,
    const CharType* last) const;
```

Parameters

maskVal
The mask value not to be matched by a character.

first
A pointer to the first character in the range to be scanned.

last
A pointer to the character immediately following the last character in the range to be scanned.

Return Value

A pointer to the first character in a range that does not match a specified mask. If no such value exists, the function returns *last*.

Remarks

The member function returns do_scan_not(`maskVal`, `first`, `last`).

Example

```
C++
```

```
// ctype_scan_not.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char *string = "Hello, my name is John!";
```

```
   const char* i = use_facet<ctype<char> > ( loc1 ).scan_not
      ( ctype_base::alpha, string, string + strlen(string) );
   cout << "First nonalpha character is \"" << *i << "\" at position: "
      << i - string << endl;
}
```

Output

```
First nonalpha character is "," at position: 5
```

ctype::tolower

Converts a character or a range of characters to lower case.

C++

```
CharType tolower(CharType ch) const;

const CharType *tolower(CharType* first, const CharType* last) const;
```

Parameters

ch
The character to be converted to lower case.

first
A pointer to the first character in the range of characters whose cases are to be converted.

last
A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the lowercase form of the parameter *ch*. If no lowercase form exists, it returns *ch*.

The second member function returns *last*.

Remarks

The first member function returns do_tolower(`ch`). The second member function returns do_tolower(`first`, `last`).

Example

C++

```
// ctype_tolower.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char string[] = "HELLO, MY NAME IS JOHN";

   use_facet<ctype<char> > ( loc1 ).tolower
      ( string, string + strlen(string) );
   cout << "The lowercase string is: " << string << endl;
}
```

Output

```
The lowercase string is: hello, my name is john
```

ctype::toupper

Converts a character or a range of characters to upper case.

C++

```
CharType toupper(CharType ch) const;
const CharType *toupper(CharType* first, const CharType* last) const;
```

Parameters

ch
The character to be converted to uppercase.

first
A pointer to the first character in the range of characters whose cases are to be

converted.

last
A pointer to the character immediately following the last character in the range of characters whose cases are to be converted.

Return Value

The first member function returns the uppercase form of the parameter *ch*. If no uppercase form exists, it returns *ch*.

The second member function returns *last*.

Remarks

The first member function returns do_toupper(`ch`). The second member function returns do_toupper(`first`, `last`).

Example

C++

```
// ctype_toupper.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "German_Germany" );

   char string[] = "Hello, my name is John";

   use_facet<ctype<char> > ( loc1 ).toupper
      ( string, string + strlen(string) );
   cout << "The uppercase string is: " << string << endl;
}
```

Output

```
The uppercase string is: HELLO, MY NAME IS JOHN
```

ctype::widen

Converts a character of type `char` in the native character set to the corresponding character of type `CharType` used by a locale.

```
C++
```

```
CharType widen(char byte) const;
const char *widen(const char* first, const char* last, CharType* dest)
const;
```

Parameters

byte
The character of type char in the native character set to be converted.

first
A pointer to the first character in the range of characters to be converted.

last
A pointer to the character immediately following the last character in the range of characters to be converted.

dest
A pointer to the first character of type `CharType` in the destination range that stores the converted range of characters.

Return Value

The first member function returns the character of type `CharType` that corresponds to the parameter character of native type `char`.

The second member function returns a pointer to the destination range of characters of type `CharType` used by a locale converted from native characters of type `char`.

Remarks

The first member function returns do_widen(`byte`). The second member function returns do_widen(`first`, `last`, `dest`).

Example

```
C++
```

```cpp
// ctype_widen.cpp
// compile with: /EHsc /W3
#include <locale>
#include <iostream>
using namespace std;

int main( )
{
   locale loc1 ( "English" );
   char *str1 = "Hello everyone!";
   wchar_t str2 [16];
   bool result1 = (use_facet<ctype<wchar_t> > ( loc1 ).widen
      ( str1, str1 + strlen(str1), &str2[0] ) != 0);  // C4996
   str2[strlen(str1)] = '\0';
   cout << str1 << endl;
   wcout << &str2[0] << endl;

   ctype<wchar_t>::char_type charT;
   charT = use_facet<ctype<char> > ( loc1 ).widen( 'a' );
}
```

Output

```
Hello everyone!
Hello everyone!
```

See also

<locale>

Thread Safety in the C++ Standard Library

`ctype<char>` Class

Article • 12/06/2021

The class is an explicit specialization of class template `ctype\<CharType>` to type **`char`**, describing an object that can serve as a locale facet to characterize various properties of a character of type **`char`**.

Syntax

C++

```
template <>
class ctype<char>
: public ctype_base
{
public:
    typedef char _Elem;
    typedef _Elem char_type;
    bool is(
    mask _Maskval,
    _Elem _Ch) const;

    const _Elem* is(
    const _Elem* first,
    const _Elem* last,
    mask* dest) const;

    const _Elem* scan_is(
    mask _Maskval,
    const _Elem* first,
    const _Elem* last) const;

    const _Elem* scan_not(
    mask _Maskval,
    const _Elem* first,
    const _Elem* last) const;

    _Elem tolower(
    _Elem _Ch) const;

    const _Elem* tolower(
    _Elem* first,
    const _Elem* last) const;

    _Elem toupper(
    _Elem _Ch) const;

    const _Elem* toupper(
    _Elem* first,
```

```
    const _Elem* last) const;

    _Elem widen(
    char _Byte) const;

    const _Elem* widen(
    const char* first,
    const char* last,
    _Elem* dest) const;

    const _Elem* _Widen_s(
    const char* first,
    const char* last,
    _Elem* dest,
    size_t dest_size) const;

    _Elem narrow(
    _Elem _Ch,
    char _Dflt = '\0') const;

    const _Elem* narrow(
    const _Elem* first,
    const _Elem* last,
    char _Dflt,
    char* dest) const;

    const _Elem* _Narrow_s(
    const _Elem* first,
    const _Elem* last,
    char _Dflt,
    char* dest,
    size_t dest_size) const;

    static locale::id& id;
    explicit ctype(
    const mask* _Table = 0,
    bool _Deletetable = false,
    size_t _Refs = 0);

protected:
    virtual ~ctype();
//other protected members
};
```

Remarks

The explicit specialization differs from the class template in several ways:

- An object of class `ctype<char>` stores a pointer to the first element of a ctype mask table, an array of UCHAR_MAX + 1 elements of type `ctype_base::mask`. It also

stores a Boolean object that indicates whether the array should be deleted (using `operator delete[]`) when the ctype< **Elem**> object is destroyed.

- Its sole public constructor lets you specify `tab`, the ctype mask table, and `del`, the Boolean object that is true if the array should be deleted when the `ctype<char>` object is destroyed, as well as the reference-count parameter refs.
- The protected member function `table` returns the stored ctype mask table.
- The static member object `table_size` specifies the minimum number of elements in a ctype mask table.
- The protected static member function `classic_table`(returns the ctype mask table appropriate to the "C" locale.
- There are no protected virtual member functions do_is, do_scan_is, or do_scan_not. The corresponding public member functions perform the equivalent operations themselves.

The member functions do_narrow and do_widen copy elements unaltered.

Requirements

Header: <locale>

Namespace: std

See also

facet Class
ctype_base Class
Thread Safety in the C++ Standard Library

ctype_base Class

Article • 05/20/2022

The class serves as a base class for facets of class template ctype. A base class for the ctype class that is used to define enumeration types used to classify or test characters either individually or within entire ranges.

Syntax

C++

```
struct ctype_base : public locale::facet
{
    enum
    {
        alnum,
        alpha,
        cntrl,
        digit,
        graph,
        lower,
        print,
        punct,
        space,
        upper,
        xdigit
    };
    typedef short mask;

    ctype_base( size_t _Refs = 0 );
    ~ctype_base();
};
```

Remarks

It defines an enumeration mask. Each enumeration constant characterizes a different way to classify characters, as defined by the functions with similar names declared in the header <ctype.h>. The constants are:

- **space** (function isspace)
- **print** (function isprint)
- **cntrl** (function iscntrl)

- **upper** (function isupper)
- **lower** (function islower)
- **digit** (function isdigit)
- **punct** (function ispunct)
- **xdigit** (function isxdigit)
- **alpha** (function isalpha)
- **alnum** (function isalnum)
- **graph** (function isgraph)

You can characterize a combination of classifications by a bitwise OR of these constants. In particular, it's always true that **alnum** == (**alpha** | **digit**) and **graph** == (**alnum** | **punct**).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

ctype_byname Class

Article • 08/03/2021

The derived class template describes an object that can serve as a ctype facet of a given locale, enabling the classification of characters and conversion of characters between case and native and locale specified character sets.

Syntax

C++

```
template <class _Elem>
class ctype_byname : public ctype<_Elem>
{
public:
    explicit ctype_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit ctype_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual __CLR_OR_THIS_CALL ~ctype_byname();

};
```

Remarks

Its behavior is determined by the named locale `_Locname`. Each constructor initializes its base object with ctype<CharType>(`_Refs`) or the equivalent for base class `ctype<char>`.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

locale Class

Article • 06/17/2022

The class that describes a locale object that encapsulates culture-specific information as a set of facets that collectively define a specific localized environment.

Syntax

```
C++

class locale;
```

Remarks

A facet is a pointer to an object of a class derived from class facet that has a public object of the form:

```
C++

static locale::id id;
```

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the locale categories traditionally managed in the Standard C Library by the function `setlocale`.

Category `collate` (LC_COLLATE) includes the facets:

```
C++

collate<char>
collate<wchar_t>
```

Category `ctype` (LC_CTYPE) includes the facets:

```
C++

ctype<char>
ctype<wchar_t>
codecvt<char, char, mbstate_t>
codecvt<wchar_t, char, mbstate_t>
```

```
codecvt<char16_t, char, mbstate_t>
codecvt<char32_t, char, mbstate_t>
```

Category `monetary` (LC_MONETARY) includes the facets:

```
C++

moneypunct<char, false>
moneypunct<wchar_t, false>
moneypunct<char, true>
moneypunct<wchar_t, true>
money_get<char, istreambuf_iterator<char>>
money_get<wchar_t, istreambuf_iterator<wchar_t>>
money_put<char, ostreambuf_iterator<char>>
money_put<wchar_t, ostreambuf_iterator<wchar_t>>
```

Category `numeric` (LC_NUMERIC) includes the facets:

```
C++

num_get<char, istreambuf_iterator<char>>
num_get<wchar_t, istreambuf_iterator<wchar_t>>
num_put<char, ostreambuf_iterator<char>>
num_put<wchar_t, ostreambuf_iterator<wchar_t>>
numpunct<char>
numpunct<wchar_t>
```

Category `time` (LC_TIME) includes the facets:

```
C++

time_get<char, istreambuf_iterator<char>>
time_get<wchar_t, istreambuf_iterator<wchar_t>>
time_put<char, ostreambuf_iterator<char>>
time_put<wchar_t, ostreambuf_iterator<wchar_t>>
```

Category `messages` (LC_MESSAGES) includes the facets:

```
C++

messages<char>
messages<wchar_t>
```

(The last category is required by POSIX, but not the C Standard.)

Some of these predefined facets are used by the `iostream` classes, to control the conversion of numeric values to and from text sequences.

An object of class locale also stores a locale name as an object of class string. Using an invalid locale name to construct a locale facet or a locale object throws an object of class runtime_error. The stored locale name is `"*"` if the locale object can't be certain that a C-style locale corresponds exactly to the one represented by the object. Otherwise, you can establish a matching locale within the Standard C Library, for some locale object `locale_object`, by calling `setlocale(LC_ALL , locale_object.name().c_str())`.

In this implementation, you can also call the static member function:

```
C++

static locale empty();
```

to construct a locale object that has no facets. It's also a transparent locale. If the template functions has_facet and use_facet can't find the requested facet in a transparent locale, they consult first the global locale and then, if that is transparent, the classic locale. So, you can write:

```
C++

cout.imbue(locale::empty());
```

Subsequent insertions to cout are mediated by the current state of the global locale. You can even write:

```
C++

locale loc(locale::empty(),
    locale::classic(),
    locale::numeric);

cout.imbue(loc);
```

Numeric formatting rules for subsequent insertions to `cout` remain the same as in the C locale, even as the global locale supplies changing rules for inserting dates and monetary amounts.

Constructors

Constructor	Description
locale	Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale.

Typedefs

Type name	Description
category	An integer type that provides bitmask values to denote standard facet families.

Member functions

Member function	Description
combine	Inserts a facet from a specified locale into a target locale.
name	Returns the stored locale name.

Static Functions

Name	Description
classic	The static member function returns a locale object that represents the classic C locale.
global	Resets the default local for the program.

Operators

Operator	Description
operator=	Assigns a locale.
operator!=	Tests two locales for inequality.
operator( )	Compares two `basic_string` objects.
operator==	Tests two locales for equality.

Classes

Class	Description

Class	Description
facet	A class that serves as the base class for all locale facets.
id	The member class provides a unique facet identification used as an index for looking up facets in a locale.

Requirements

Header: <locale>

Namespace: std

locale::category

An integer type that provides bitmask values to denote standard facet families.

```
C++
```

```
typedef int category;
static const int collate = LC_COLLATE;
static const int ctype = LC_CTYPE;
static const int monetary = LC_MONETARY;
static const int numeric = LC_NUMERIC;
static const int time = LC_TIME;
static const int messages = LC_MESSAGES;
static const int all = LC_ALL;
static const int none = 0;
```

Remarks

The type is a synonym for an `int` type that can represent a group of distinct elements of a bitmask type local to class locale or can be used to represent any of the corresponding C locale categories. The elements are:

- `collate`, corresponding to the C category LC_COLLATE
- `ctype`, corresponding to the C category LC_CTYPE
- `monetary`, corresponding to the C category LC_MONETARY
- `numeric`, corresponding to the C category LC_NUMERIC
- `time`, corresponding to the C category LC_TIME

- `messages`, corresponding to the POSIX category LC_MESSAGES

Two more useful values are:

- `none`, corresponding to none of the C categories
- `all`, corresponding to the C union of all categories LC_ALL

You can represent an arbitrary group of categories by using a bitwise-OR with these constants, as in `monetary | time`.

locale::classic

The static member function returns a locale object that represents the classic C locale.

C++

```
static const locale& classic();
```

Return Value

A reference to the C locale.

Remarks

The classic C locale is the U.S. English ASCII locale within the Standard C library. It's the locale that's used implicitly in programs that aren't internationalized.

Example

C++

```
// locale_classic.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "german" );
   locale loc2 = locale::global( loc1 );
   cout << "The name of the previous locale is: " << loc2.name( )
```

```
        << "." << endl;
   cout << "The name of the current locale is: " << loc1.name( )
        << "." << endl;

   if (loc2 == locale::classic( ) )
      cout << "The previous locale was classic." << endl;
   else
      cout << "The previous locale was not classic." << endl;

   if (loc1 == locale::classic( ) )
      cout << "The current locale is classic." << endl;
   else
      cout << "The current locale is not classic." << endl;
}
```

Output

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
The previous locale was classic.
The current locale is not classic.
```

locale::combine

Inserts a facet from a specified locale into a target locale.

C++

```
template <class Facet>
locale combine(const locale& source_locale) const;
```

Parameters

source_locale
The locale containing the facet to be inserted into the target locale.

Return Value

The member function returns a locale object that replaces in or adds to `*this` the facet `Facet` listed in *source_locale*.

Example

C++

```
// locale_combine.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main() {
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-
s; it comes before z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

   locale loc2 ( "C" );
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc2 )  << result2 << endl;

   locale loc3 = loc2.combine<collate<_TCHAR> > (loc);
   int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;
}
```

facet Class

A class that serves as the base class for all locale facets.

C++

```
class facet {
protected:
    explicit facet(size_t references = 0);
    virtual ~facet();
private:
    facet(const facet&) // not defined
    void operator=(const facet&) // not defined
};
```

Remarks

You can't copy or assign an object of class `facet`. You can construct and destroy objects derived from class `locale::facet` but not objects of the base class proper. Typically, you

construct an object `_Myfac` derived from `facet` when you construct a `locale`, as in `locale loc(locale::classic(), new _Myfac);`

In such cases, the constructor for the base class `facet` should have a zero *references* argument. When the object is no longer needed, it's deleted. Supply a nonzero *references* argument only in those rare cases where you take responsibility for the lifetime of the object.

locale::global

Resets the default locale for the program. This call affects the global locale for both C and C++.

C++

```
static locale global(const locale& new_default_locale);
```

Parameters

new_default_locale
The locale to be used as the default locale by the program.

Return Value

The previous locale before the default locale was reset.

Remarks

At program startup, the global locale is the same as the classic locale. The `global()` function calls `setlocale( LC_ALL, loc.name. c_str())` to establish a matching locale in the Standard C library.

Example

C++

```
// locale_global.cpp
// compile by using: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
```

```
using namespace std;

int main( )
{
   locale loc ( "German_germany" );
   locale loc1;
   cout << "The initial locale is: " << loc1.name( ) << endl;
   locale loc2 = locale::global ( loc );
   locale loc3;
   cout << "The current locale is: " << loc3.name( ) << endl;
   cout << "The previous locale was: " << loc2.name( ) << endl;
}
```

Output

```
The initial locale is: C
The current locale is: German_Germany.1252
The previous locale was: C
```

id Class

The member class provides a unique facet identification used as an index for looking up facets in a locale.

C++

```
class id
{
   protected:    id();
   private:      id(const id&)
   void operator=(const id&)  // not defined
};
```

Remarks

The member class describes the static member object required by each unique locale facet. You can't copy or assign an object of class `id`.

locale::locale

Creates a locale, or a copy of a locale, or a copy of locale where a facet or a category has been replaced by a facet or category from another locale. Also includes a destructor.

C++

```
locale();

explicit locale(const char* locale_name, category new_category = all);
explicit locale(const string& locale_name);
locale(const locale& from_locale);
locale(const locale& from_locale, const locale& Other, category
new_category);
locale(const locale& from_locale, const char* locale_name, category
new_category);

template <class Facet>
locale(const locale& from_locale, const Facet* new_facet);

~locale();
```

Parameters

locale_name
Name of a locale.

from_locale
A locale that is to be copied in constructing the new locale.

Other
A locale from which to select a category.

new_category
The category to be substituted into the constructed locale.

new_facet
The facet to be substituted into the constructed locale.

Remarks

The first constructor initializes the object to match the global locale. The second and third constructors initialize all the locale categories to have behavior consistent with the locale name *locale_name*. The remaining constructors copy *from_locale*, with the exceptions noted:

`locale(const locale& from_locale, const locale& Other, category new_category);`

replaces from *Other* those facets corresponding to a category C for which C & *new_category* is nonzero.

`locale(const locale& from_locale, const char* locale_name, category new_category);`

`locale(const locale& from_locale, const string& locale_name, category new_category);`

replaces from `locale(locale_name, all)` those facets corresponding to a category *replace_category* for which `replace_category & new_category` is nonzero.

`template<class Facet> locale(const locale& from_locale, Facet* new_facet);`

replaces in (or adds to) *from_locale* the facet *new_facet*, if *new_facet* isn't a null pointer.

If locale name *locale_name* is a null pointer or otherwise invalid, the function throws runtime_error.

Example

C++

```
// locale_locale.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;

int main( ) {

   // Second constructor
   locale loc ( "German_germany" );
   _TCHAR * s1 = _T("Das ist wei\x00dfzz."); // \x00df is the German sharp-
s, it comes before z in the German alphabet
   _TCHAR * s2 = _T("Das ist weizzz.");
   int result1 = use_facet<collate<_TCHAR> > ( loc ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc ) << result1 << endl;

   // The first (default) constructor
   locale loc2;
   int result2 = use_facet<collate<_TCHAR> > ( loc2 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc2 )  << result2 << endl;

   // Third constructor
   locale loc3 (loc2,loc, _M_COLLATE );
   int result3 = use_facet<collate<_TCHAR> > ( loc3 ).
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc3 ) << result3 << endl;

   // Fourth constructor
   locale loc4 (loc2, "German_Germany", _M_COLLATE );
   int result4 = use_facet<collate<_TCHAR> > ( loc4 ).
```

```
      compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
   cout << isalpha (_T ( '\x00df' ), loc4 ) << result4 << endl;
}
```

locale::name

Returns the stored locale name.

C++

```
string name() const;
```

Return Value

A string giving the name of the locale.

Example

C++

```
// locale_name.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "german" );
   locale loc2 = locale::global( loc1 );
   cout << "The name of the previous locale is: "
        << loc2.name( ) << "." << endl;
   cout << "The name of the current locale is: "
        << loc1.name( ) << "." << endl;
}
```

Output

```
The name of the previous locale is: C.
The name of the current locale is: German_Germany.1252.
```

locale::operator=

Assigns a locale.

```
C++
```

```
const locale& operator=(const locale& other) noexcept;
```

locale::operator!=

Tests two locales for inequality.

```
C++
```

```
bool operator!=(const locale& right) const;
```

Parameters

right
One of the locales to be tested for inequality.

Return Value

A Boolean value that is `true` if the locales aren't copies of the same locale. It's `false` if the locales are copies of the same locale.

Remarks

Two locales are equal if they're the same locale, if one is a copy of the other, or if they have identical names.

Example

```
C++
```

```
// locale_op_ne.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;
```

```
int main( )
{
   locale loc1( "German_Germany" );
   locale loc2( "German_Germany" );
   locale loc3( "English" );

   if ( loc1 != loc2 )
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc2 (" << loc2.name( ) << ") are not equal." << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc2 (" << loc2.name( ) << ") are equal." << endl;

   if ( loc1 != loc3 )
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc3 (" << loc3.name( ) << ") are not equal." << endl;
   else
      cout << "locales loc1 (" << loc1.name( )
      << ") and\n loc3 (" << loc3.name( ) << ") are equal." << endl;
}
```

Output

```
locales loc1 (German_Germany.1252) and
loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252) and
loc3 (English_United States.1252) are not equal.
```

locale::operator()

Compares two `basic_string` objects according to the lexicographic comparison rules defined by this locale's `std::collate<charT>` facet.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator()(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right) const;
```

Parameters

left

The first string to compare.

right
The second string to compare.

Return Value

- `true` if *left* is lexicographically less than *right*, otherwise `false`.

Remarks

The member function effectively executes:

C++

```
const collate<CharType>& fac = use_fac<collate<CharType>>(*this);

return (fac.compare(left.begin(), left.end(), right.begin(), right.end()) <
0);
```

It means you can use a locale object as a function object.

Example

C++

```
// locale_op_compare.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

int main( )
{
   using namespace std;
   const wchar_t *sa = L"ztesting";
   const wchar_t *sb = L"\0x00DFtesting";
   basic_string<wchar_t> a( sa );
   basic_string<wchar_t> b( sb );

   locale loc( "German_Germany" );
   cout << loc( a,b ) << endl;

   const collate<wchar_t>& fac = use_facet<collate<wchar_t> >( loc );
   cout << ( fac.compare( sa, sa + a.length( ),
       sb, sb + b.length( ) ) < 0) << endl;
}
```

Output

```
0
0
```

locale::operator==

Tests two locales for equality.

C++

```
bool operator==(const locale& right) const;
```

Parameters

right
One of the locales to be tested for equality.

Return Value

A Boolean value that is `true` if the locales are copies of the same locale. It's `false` if the locales aren't copies of the same locale.

Remarks

Two locales are equal if they're the same locale, if one is a copy of the other, or if they have identical names.

Example

C++

```
// locale_op_eq.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
#include <locale>

using namespace std;

int main( )
{
   locale loc1( "German_Germany" );
```

```
    locale loc2( "German_Germany" );
    locale loc3( "English" );

    if ( loc1 == loc2 )
       cout << "locales loc1 (" << loc1.name( )
       << ")\n and loc2 (" << loc2.name( ) << ") are equal."
       << endl;
    else
       cout << "locales loc1 (" << loc1.name( )
       << ")\n and loc2 (" << loc2.name( ) << ") are not equal."
       << endl;

    if ( loc1 == loc3 )
       cout << "locales loc1 (" << loc1.name( )
       << ")\n and loc3 (" << loc3.name( ) << ") are equal."
       << endl;
    else
       cout << "locales loc1 (" << loc1.name( )
       << ")\n and loc3 (" << loc3.name( ) << ") are not equal."
       << endl;
}
```

Output

```
locales loc1 (German_Germany.1252)
and loc2 (German_Germany.1252) are equal.
locales loc1 (German_Germany.1252)
and loc3 (English_United States.1252) are not equal.
```

See also

<locale>

Code Pages

Locale Names, Languages, and Country/Region Strings

Thread Safety in the C++ Standard Library

messages Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to retrieve localized messages from a catalog of internationalized messages for a given locale.

Currently, while the messages class is implemented, there are no messages.

Syntax

C++

```
template <class CharType>
class messages : public messages_base;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

This facet basically opens a catalog of messages defined in the base class messages_base, retrieves the information required, and closes the catalog.

Constructors

Constructor	Description
messages	The message facet constructor function.

Typedefs

Type name	Description

Type name	Description
char_type	A character type that is used display messages.
string_type	A type that describes a string of type `basic_string` containing characters of type `CharType`.

Member functions

Member function	Description
close	Closes the message catalog.
do_close	A virtual function called to lose the message catalog.
do_get	A virtual function called to retrieve the message catalog.
do_open	A virtual function called to open the message catalog.
get	Retrieves the message catalog.
open	Opens the message catalog.

Requirements

Header: <locale>

Namespace: std

messages::char_type

A character type that is used display messages.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

messages::close

Closes the message catalog.

```
C++
```

```
void close(catalog _Catval) const;
```

Parameters

_Catval
The catalog to be closed.

Remarks

The member function calls do_close(_ *Catval*).

messages::do_close

A virtual function called to lose the message catalog.

```
C++
```

```
virtual void do_close(catalog _Catval) const;
```

Parameters

_Catval
The catalog to be closed.

Remarks

The protected member function closes the message catalog *_Catval*, which must have been opened by an earlier call to do_open.

_Catval must be obtained from a previously opened catalog that is not closed.

Example

See the example for close, which calls `do_close`.

messages::do_get

A virtual function called to retrieve the message catalog.

```
C++
```

```
virtual string_type do_get(
    catalog _Catval,
    int _Set,
    int _Message,
    const string_type& _Dfault) const;
```

Parameters

_Catval
The identification value specifying the message catalog to be searched.

_Set
The first identified used to locate a message in a message catalog.

_Message
The second identified used to locate a message in a message catalog.

_Dfault
The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The protected member function tries to obtain a message sequence from the message catalog *_Catval*. It may make use of *_Set*, *_Message*, and *_Dfault* in doing so.

Example

See the example for get, which calls `do_get`.

messages::do_open

A virtual function called to open the message catalog.

```
C++
```

```
virtual catalog do_open(
    const string& _Catname,
    const locale& _Loc) const;
```

Parameters

_Catname
The name of the catalog to be searched.

_Loc
The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to get.

Remarks

The protected member function tries to open a message catalog whose name is *_Catname*. It may make use of the locale *_Loc* in doing so

The return value should be used as the argument on a later call to close.

Example

See the example for open, which calls `do_open`.

messages::get

Retrieves the message catalog.

```
C++
```

```
string_type get(
    catalog _CatVal,
    int _Set,
    int _Message,
    const string_type& _Dfault) const;
```

Parameters

_Catval
The identification value specifying the message catalog to be searched.

_Set
The first identified used to locate a message in a message catalog.

_Message
The second identified used to locate a message in a message catalog.

_Dfault
The string to be returned on failure.

Return Value

It returns a copy of *_Dfault* on failure. Otherwise, it returns a copy of the specified message sequence.

Remarks

The member function returns do_get(`_Catval`, `_Set`, `_Message`, `_Dfault`).

messages::messages

The message facet constructor function.

```
C++
```

```
explicit messages(
    size_t _Refs = 0);

protected: messages(
    const char* _Locname,
    size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

_Locname
The name of the locale.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(`_Refs`).

messages::open

Opens the message catalog.

```
C++
```

```
catalog open(
    const string& _Catname,
    const locale& _Loc) const;
```

Parameters

_Catname
The name of the catalog to be searched.

_Loc
The locale being searched for in the catalog.

Return Value

It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to get.

Remarks

The member function returns do_open(`_Catname`, `_Loc`).

messages::string_type

A type that describes a string of type `basic_string` containing characters of type `CharType`.

```
C++
```

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of class template basic_string whose objects can store copies of the message sequences.

See also

<locale>
messages_base Class
Thread Safety in the C++ Standard Library

messages_base Class

Article • 08/03/2021

The base class describes an `int` type for the catalog of messages.

Syntax

```
C++
```

```
struct messages_base : locale::facet {
    typedef int catalog;
    explicit messages_base(size_t _Refs = 0)
};
```

Remarks

The type catalog is a synonym for type `int` that describes the possible return values from messages:: do_open.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

messages_byname Class

Article • 08/03/2021

The derived class template describes an object that can serve as a message facet of a given locale, enabling the retrieval of localized messages.

Syntax

```
C++
```

```
template <class CharType>
class messages_byname : public messages<CharType> {
public:
    explicit messages_byname(
    const char *_Locname,
    size_t _Refs = 0);

    explicit messages_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~messages_byname();

};
```

Parameters

_Locname
A named locale.

_Refs
An initial reference count.

Remarks

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with messages<CharType>(`_Refs`).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

money_base Class

Article • 08/03/2021

The class describes an enumeration and a structure common to all specializations of class template moneypunct.

Syntax

```
C++
```

```
struct pattern
{
   char field[_PATTERN_FIELD_SIZE];
};
```

Remarks

The enumeration `part` describes the possible values in elements of the array field in the structure pattern. The values of `part` are:

- `none` to match zero or more spaces or generate nothing.
- `sign` to match or generate a positive or negative sign.
- `space` to match zero or more spaces or generate a space.
- `symbol` to match or generate a currency symbol.
- `value` to match or generate a monetary value.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

money_get Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to monetary values.

Syntax

```
C++
```

```
template <class CharType, class InputIterator =
istreambuf_iterator<CharType>>
class money_get : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

InputIterator
The type of iterator from which the get functions read their input.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

Constructor	Description
money_get	The constructor for objects of type `money_get` that are used to extract numerical values from sequences representing monetary values.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.

Type name	Description
iter_type	A type that describes an input iterator.
string_type	A type that describes a string containing characters of type `CharType`.

Member functions

Member function	Description
do_get	A virtual function called to extracts a numerical value from a character sequence that represents a monetary value.
get	Extracts a numerical value from a character sequence that represents a monetary value.

Requirements

Header: <locale>

Namespace: std

money_get::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter *CharType*.

money_get::do_get

Virtual function called to extracts a numerical value from a character sequence that represents a monetary value.

```
C++
```

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
    ios_base::iostate& State,
    long double& val) const virtual iter_type do_get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
    ios_base::iostate& State,
    string_type& val) const
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

Intl
A Boolean value indicating the type of currency symbol expected in the sequence: `true` if international, `false` if domestic.

Iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

State
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded or not.

val
A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

The first virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete,

nonempty monetary input field. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (`-`), to represent the amount and stores the result in the string_type object *val*. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in *val* and sets `ios_base::failbit` in *State*. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `State`.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally signed digit sequence to a value of type `long double` and stores that value in *val*.

The format of a monetary input field is determined by the locale facet**fac** returned by the effective call use_facet < moneypunct< **CharType**, **intl**>>(**iosbase**. getloc).

Specifically:

- **fac**. neg_format determines the order in which components of the field occur.
- **fac**. curr_symbol determines the sequence of elements that constitutes a currency symbol.
- **fac**. positive_sign determines the sequence of elements that constitutes a positive sign.
- **fac**. negative_sign determines the sequence of elements that constitutes a negative sign.
- **fac**. grouping determines how digits are grouped to the left of any decimal point.
- **fac**. thousands_sep determines the element that separates groups of digits to the left of any decimal point.
- **fac**. decimal_point determines the element that separates the integer digits from the fraction digits.
- **fac**. frac_digits determines the number of significant fraction digits to the right of any decimal point. When parsing a monetary amount with more fraction digits than are called for by `frac_digits`, `do_get` stops parsing after consuming at most `frac_digits` characters.

If the sign string (**fac**. `negative_sign` or **fac**. `positive_sign`) has more than one element, only the first element is matched where the element equal to **money_base::sign** appears

in the format pattern (**fac**. `neg_format`). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If **iosbase**. flags & showbase is nonzero, the string **fac**. `curr_symbol` must match where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, if **money_base::symbol** occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is not matched. Otherwise, the currency symbol is optionally matched.

If no instances of **fac**. `thousands_sep` occur in the value portion of the monetary input field (where the element equal to **money_base::value** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by **fac**. **grouping** is enforced. Note that the resulting digit sequence represents an integer whose low-order **fac**. `frac_digits` decimal digits are considered to the right of the decimal point.

Arbitrary white space is matched where the element equal to **money_base::space** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element *ch* is considered white space if use_facet < ctype< **CharType**> >(**iosbase**. getloc). is(**ctype_base::space**, *ch*) is `true`.

Example

See the example for get, which calls `do_get`.

money_get::get

Extracts a numerical value from a character sequence that represents a monetary value.

C++

```
iter_type get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
    ios_base::iostate& State,
    long double& val) const;

iter_type get(iter_type first,
    iter_type last,
    bool Intl,
    ios_base& Iosbase,
```

```
    ios_base::iostate& State,
    string_type& val) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

Intl
A Boolean value indicating the type of currency symbol expected in the sequence: `true` if international, `false` if domestic.

Iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

State
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

val
A string storing the converted sequence.

Return Value

An input iterator addressing the first element beyond the monetary input field.

Remarks

Both member functions return do_get`(first, last, Intl, Iosbase, State, val)`.

Example

C++

```
// money_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
```

```
using namespace std;

int main( )
{
   locale loc( "german_germany" );

   basic_stringstream< char > psz;
   psz << use_facet<moneypunct<char, 1> >(loc).curr_symbol() << "-1.000,56";
   basic_stringstream< char > psz2;
   psz2 << "-100056" << use_facet<moneypunct<char, 1> >(loc).curr_symbol();

   ios_base::iostate st = 0;
   long double fVal;

   psz.flags( psz.flags( )|ios_base::showbase );
   // Which forced the READING the currency symbol
   psz.imbue(loc);
   use_facet < money_get < char > >( loc ).
      get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
           basic_istream<char>::_Iter( 0 ), true, psz, st, fVal );

   if ( st & ios_base::failbit )
      cout << "money_get(" << psz.str( ) << ", intl = 1) FAILED"
           << endl;
   else
      cout << "money_get(" << psz.str( ) << ", intl = 1) = "
           << fVal/100.0 << endl;

   use_facet < money_get < char > >( loc ).
      get(basic_istream<char>::_Iter(psz2.rdbuf( )),
          basic_istream<char>::_Iter(0), false, psz2, st, fVal);

   if ( st & ios_base::failbit )
      cout << "money_get(" << psz2.str( ) << ", intl = 0) FAILED"
           << endl;
   else
      cout << "money_get(" << psz2.str( ) << ", intl = 0) = "
           << fVal/100.0 << endl;
};
```

money_get::iter_type

A type that describes an input iterator.

C++

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

money_get::money_get

The constructor for objects of type `money_get` that are used to extract numerical values from sequences representing monetary values.

```
C++
```

```
explicit money_get(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(*_Refs*).

money_get::string_type

A type that describes a string containing characters of type **CharType**.

```
C++
```

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of class template basic_string.

See also

<locale>

facet Class

Thread Safety in the C++ Standard Library

money_put Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to control conversions of monetary values to sequences of type `CharType`.

Syntax

C++

```
template <class CharType,
    class OutputIterator = ostreambuf_iterator<CharType>>
class money_put : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

OutputIterator
The type of iterator to which the monetary put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

Constructor	Description
money_put	The constructor for objects of type `money_put`.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.

Type name	Description
string_type	A type that describes a string containing characters of type `CharType`.

Member functions

Member function	Description
do_put	A virtual function called to convert either number or a string to a character sequence that represents a monetary value.
put	Converts either number or a string to a character sequence that represents a monetary value.

Requirements

Header: <locale>

Namespace: std

money_put::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

money_put::do_put

A virtual function called to convert either number or a string to a character sequence that represents a monetary value.

```
C++

virtual iter_type do_put(
    iter_type next,
```

```
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    const string_type& val) const;

virtual iter_type do_put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

Parameters

next
An iterator addressing the first element of the inserted string.

_Intl
A Boolean value indicating the type of currency symbol expected in the sequence: `true` if international, `false` if domestic.

_Iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill
A character which is used for spacing.

val
A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *next* to produce a monetary output field from the string_type object *val*. The sequence controlled by *val* must begin with one or more decimal digits, optionally preceded by a minus sign (-), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts *val* to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the locale facet fac returned by the (effective) call use_facet < moneypunct< **CharType**, **intl**> >(**iosbase**. getloc).

Specifically:

- **fac**. pos_format determines the order in which components of the field are generated for a nonnegative value.
- **fac**. neg_format determines the order in which components of the field are generated for a negative value.
- **fac**. curr_symbol determines the sequence of elements to generate for a currency symbol.
- **fac**. positive_sign determines the sequence of elements to generate for a positive sign.
- **fac**. negative_sign determines the sequence of elements to generate for a negative sign.
- **fac**. grouping determines how digits are grouped to the left of any decimal point.
- **fac**. thousands_sep determines the element that separates groups of digits to the left of any decimal point.
- **fac**. decimal_point determines the element that separates the integer digits from any fraction digits.
- **fac**. frac_digits determines the number of significant fraction digits to the right of any decimal point.

If the sign string (**fac**. `negative_sign` or **fac**. `positive_sign`) has more than one element, only the first element is generated where the element equal to **money_base::sign** appears in the format pattern (**fac**. `neg_format` or **fac**. `pos_format`). Any remaining elements are generated at the end of the monetary output field.

If **iosbase**. flags & showbase is nonzero, the string **fac**. `curr_symbol` is generated where the element equal to **money_base::symbol** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by **fac**. **grouping** (its first element has the value CHAR_MAX), then no instances of **fac**. `thousands_sep` are generated in the value portion of the monetary output field (where the element equal to **money_base::value** appears in the format pattern). If **fac**. `frac_digits` is zero, then no instance of **fac**. `decimal_point` is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order **fac**. `frac_digits` decimal digits to the right of the decimal point.

Padding occurs as for any numeric output field, except that if **iosbase**. **flags** & **iosbase**. internal is nonzero, any internal padding is generated where the element equal to **money_base::space** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is **fill**.

The function calls **iosbase**. **width**(0) to reset the field width to zero.

Example

See the example for put, where the virtual member function is called by **put**.

money_put::iter_type

A type that describes an output iterator.

```
C++

typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator.**

money_put::money_put

The constructor for objects of type `money_put`.

```
C++

explicit money_put(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: the lifetime of the object is managed by the locales that contain it.
- 1: the lifetime of the object must be manually managed.
- > 1: these values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(`_Refs`).

money_put::put

Converts either number or a string to a character sequence that represents a monetary value.

```
C++
```

```
iter_type put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    const string_type& val) const;

iter_type put(
    iter_type next,
    bool _Intl,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

Parameters

next
An iterator addressing the first element of the inserted string.

_Intl
A Boolean value indicating the type of currency symbol expected in the sequence: `true`

if international, `false` if domestic.

_Iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required

_Fill
A character which is used for spacing.

val
A string object to be converted.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

Both member functions return do_put(`next`, `_Intl`, `_Iosbase`, `_Fill`, `val`).

Example

C++

```
// money_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

int main()
{
    std::locale loc( "german_germany" );
    std::basic_stringstream<char> psz;

    psz.imbue(loc);
    psz.flags(psz.flags() | std::ios_base::showbase); // force the printing
of the currency symbol
    std::use_facet<std::money_put<char> >
(loc).put(std::basic_ostream<char>::_Iter(psz.rdbuf()), true, psz, ' ',
100012);
    if (psz.fail())
        std::cout << "money_put() FAILED" << std::endl;
    else
        std::cout << "money_put() = \"" << psz.rdbuf()->str() << "\"" <<
std::endl;
}
```

Output

```
money_put() = "EUR1.000,12"
```

money_put::string_type

A type that describes a string containing characters of type `CharType`.

C++

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of class template basic_string whose objects can store sequences of elements from the source sequence.

See also

<locale>
facet Class
Thread Safety in the C++ Standard Library

moneypunct Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to describe the sequences of type *CharType* used to represent a monetary input field or a monetary output field. If the template parameter *Intl* is *true*, international conventions are observed.

Syntax

C++

```
template <class CharType, bool Intl>
class moneypunct;
```

Parameters

CharType
The type used within a program to encode characters.

Intl
A flag specifying whether international conventions are to be observed.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

The const static object intl stores the value of the template parameter *Intl*.

Constructors

Constructor	Description
moneypunct	Constructor of objects of type `moneypunct`.

Typedefs

Type name	Description

Type name	Description
char_type	A type that is used to describe a character used by a locale.
string_type	A type that describes a string containing characters of type `CharType`.

Member functions

Member function	Description
curr_symbol	Returns a locale-specific sequence of elements to use as a currency symbol.
decimal_point	Returns a locale-specific sequence of elements to use as a decimal point symbol.
do_curr_symbol	A protected virtual member function that returns a locale-specific sequence of elements to use as a currency symbol.
do_decimal_point	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a decimal point symbol.
do_frac_digits	The protected virtual member function returns a locale-specific count of the number of digits to display to the right of any decimal point.
do_grouping	The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
do_neg_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs with negative amounts.
do_negative_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.
do_pos_format	A protected virtual member function that is called to return a locale-specific rule for formatting outputs with positive amounts.
do_positive_sign	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a positive sign symbol.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific sequence of elements to use as a thousands separator symbol.
frac_digits	Returns a locale-specific count of the number of digits to display to the right of any decimal point.
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
neg_format	Returns a locale-specific rule for formatting outputs with negative amounts.

Member function	Description
negative_sign	Returns a locale-specific sequence of elements to use as a negative sign symbol.
pos_format	Returns a locale-specific rule for formatting outputs with positive amounts.
positive_sign	Returns a locale-specific sequence of elements to use as a positive sign symbol.
thousands_sep	Returns a locale-specific sequence of elements to use as a thousands separator symbol.

Requirements

Header: <locale>

Namespace: std

moneypunct::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

moneypunct::curr_symbol

Returns a locale-specific sequence of elements to use as a currency symbol.

```
C++

string_type curr_symbol() const;
```

Return Value

A string containing the currency symbol.

Remarks

The member function returns do_curr_symbol.

Example

C++

```
// moneypunct_curr_symbol.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct < char, true > &mpunct = use_facet < moneypunct < char,
true > >(loc);
   cout << loc.name( ) << " international currency symbol "<<
mpunct.curr_symbol( ) << endl;

   const moneypunct < char, false> &mpunct2 = use_facet < moneypunct < char,
false> >(loc);
   cout << loc.name( ) << " domestic currency symbol "<<
mpunct2.curr_symbol( ) << endl;
};
```

moneypunct::decimal_point

Returns a locale-specific sequence of elements to use as a decimal point symbol.

C++

```
CharType decimal_point() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Remarks

The member function returns do_decimal_point.

Example

C++

```
// moneypunct_decimal_pt.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc("german_germany");

   const moneypunct < char, true > &mpunct = use_facet
      < moneypunct < char, true > >(loc);
   cout << loc.name( ) << " international decimal point "
        << mpunct.decimal_point( ) << endl;

   const moneypunct < char, false> &mpunct2 = use_facet
      < moneypunct < char, false> >(loc);
   cout << loc.name( ) << " domestic decimal point "
        << mpunct2.decimal_point( ) << endl;
}
```

Output

```
German_Germany.1252 international decimal point ,
German_Germany.1252 domestic decimal point ,
```

moneypunct::do_curr_symbol

A protected virtual member function that returns a locale-specific sequence of elements to use as a currency symbol.

C++

```
virtual string_type do_curr_symbol() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for curr_symbol, where the virtual member function is called by `curr_symbol`.

moneypunct::do_decimal_point

A protected virtual member function that returns a locale-specific sequence of elements to use as a decimal point symbol.

```
C++
```

```
virtual CharType do_decimal_point() const;
```

Return Value

A locale-specific sequence of elements to use as a decimal point symbol.

Example

See the example for decimal_point, where the virtual member function is called by `decimal_point`.

moneypunct::do_frac_digits

A protected virtual member function that returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
C++
```

```
virtual int do_frac_digits() const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Example

See the example for frac_digits, where the virtual member function is called by `frac_digits`.

moneypunct::do_grouping

A protected virtual member function that returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
C++

virtual string do_grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Example

See the example for grouping, where the virtual member function is called by `grouping`.

moneypunct::do_neg_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with negative amounts.

```
C++

virtual pattern do_neg_format() const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a negative amount. Each of the four elements of `pattern::field` can have the values:

- `none` to match zero or more spaces or generate nothing.
- `sign` to match or generate a positive or negative sign.

- `space` to match zero or more spaces or generate a space.
- `symbol` to match or generate a currency symbol.
- `value` to match or generate a monetary value.

Components of a monetary output field are generated and components of a monetary input field are matched in the order in which these elements appear in `pattern::field`. Each of the values `sign`, `symbol`, `value`, and either `none` or `space` must appear exactly once. The value `none` must not appear first. The value `space` must not appear first or last. If `Intl` is true, the order is `symbol`, `sign`, `none`, then `value`.

The template version of `moneypunct< CharType, Intl >` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

Example

See the example for neg_format, where the virtual member function is called by `neg_format`.

moneypunct::do_negative_sign

A protected virtual member function that is called to return a locale-specific sequence of elements to use as a negative sign symbol.

```
C++
```

```
virtual string_type do_negative_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a negative sign.

Example

See the example for negative_sign, where the virtual member function is called by `negative_sign`.

moneypunct::do_pos_format

A protected virtual member function that is called to return a locale-specific rule for formatting outputs with positive amounts.

```
C++
```

```
virtual pattern do_pos_format() const;
```

Return Value

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a positive amount. (It also determines how to match the components of a monetary input field.) The encoding is the same as for do_neg_format.

The template version of `moneypunct< CharType, InputIterator >` returns `{ money_base::symbol, money_base::sign, money_base::value, money_base::none }`.

Example

See the example for pos_format, where the virtual member function is called by `pos_format`.

moneypunct::do_positive_sign

A protected virtual member function that returns a locale-specific sequence of elements to use as a positive sign.

```
C++
```

```
virtual string_type do_positive_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign.

Example

See the example for positive_sign, where the virtual member function is called by `positive_sign`.

moneypunct::do_thousands_sep

A protected virtual member function that returns a locale-specific element to use as a group separator to the left of any decimal point.

```
C++
```

```
virtual CharType do_thousands_sep() const;
```

Return Value

A locale-specific element to use as a group separator to the left of any decimal point.

Example

See the example for thousands_sep, where the virtual member function is called by `thousands_sep`.

moneypunct::frac_digits

Returns a locale-specific count of the number of digits to display to the right of any decimal point.

```
C++
```

```
int frac_digits() const;
```

Return Value

A locale-specific count of the number of digits to display to the right of any decimal point.

Remarks

The member function returns do_frac_digits.

Example

```
C++
```

```cpp
// moneypunct_frac_digits.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true> >(loc);
   for (unsigned int i = 0; i <mpunct.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

   const moneypunct <char, false> &mpunct2 =
       use_facet <moneypunct <char, false> >(loc);
   for (unsigned int i = 0; i <mpunct2.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " domestic grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct2.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}
```

Output

```
German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
to the right of the radix character: 2

German_Germany.1252 domestic grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
to the right of the radix character: 2
```

moneypunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
C++
```

```
string grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns do_grouping.

Example

```
C++
```

```
// moneypunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true> >( loc );
   for (unsigned int i = 0; i <mpunct.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(mpunct.grouping ( )[i])
           << endl;
   }
   cout << loc.name( ) << " international frac_digits\n to the right"
        << " of the radix character: "
        << mpunct.frac_digits ( ) << endl << endl;

   const moneypunct <char, false> &mpunct2 =
       use_facet <moneypunct <char, false> >( loc );
   for (unsigned int i = 0; i <mpunct2.grouping( ).length( ); i++ )
   {
      cout << loc.name( ) << " domestic grouping:\n the "
```

```
            << i <<"th group to the left of the radix character "
            << "is of size " << (int)(mpunct2.grouping ( )[i])
            << endl;
   }
   cout << loc.name( ) << " domestic frac_digits\n to the right"
        << " of the radix character: "
        << mpunct2.frac_digits ( ) << endl << endl;
}
```

Output

```
German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 international frac_digits
to the right of the radix character: 2

German_Germany.1252 domestic grouping:
the 0th group to the left of the radix character is of size 3
German_Germany.1252 domestic frac_digits
to the right of the radix character: 2
```

moneypunct::moneypunct

Constructor of objects of type `moneypunct`.

C++

```
explicit moneypunct(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with locale::facet(_ *Refs*).

moneypunct::neg_format

Returns a locale-specific rule for formatting outputs with negative amounts.

```
C++
```

```
pattern neg_format() const;
```

Return Value

A locale-specific rule for formatting outputs with negative amounts.

Remarks

The member function returns do_neg_format.

Example

```
C++
```

```
// moneypunct_neg_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( ) {
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international negative number format: "
        << mpunct.neg_format( ).field[0]
        << mpunct.neg_format( ).field[1]
        << mpunct.neg_format( ).field[2]
        << mpunct.neg_format( ).field[3] << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic negative number format: "
        << mpunct2.neg_format( ).field[0]
        << mpunct2.neg_format( ).field[1]
```

```
        << mpunct2.neg_format( ).field[2]
        << mpunct2.neg_format( ).field[3] << endl;
}
```

moneypunct::negative_sign

Returns a locale-specific sequence of elements to use as a negative sign symbol.

C++

```
string_type negative_sign() const;
```

Return Value

Returns a locale-specific sequence of elements to use as a negative sign symbol.

Remarks

The member function returns do_negative_sign.

Example

C++

```
// moneypunct_neg_sign.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true> >(loc);
   cout << loc.name( ) << " international negative sign: "
        << mpunct.negative_sign( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic negative sign: "
        << mpunct2.negative_sign( ) << endl;
```

```
   locale loc2( "French" );

   const moneypunct <char, true> &mpunct3 =
      use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international negative sign: "
        << mpunct3.negative_sign( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic negative sign: "
        << mpunct4.negative_sign( ) << endl;
};
```

Output

```
German_Germany.1252 international negative sign: -
German_Germany.1252 domestic negative sign: -
French_France.1252 international negative sign: -
French_France.1252 domestic negative sign: -
```

moneypunct::pos_format

Returns a locale-specific rule for formatting outputs with positive amounts.

C++

```
pattern pos_format() const;
```

Return Value

A locale-specific rule for formatting outputs with positive amounts.

Remarks

The member function returns do_pos_format.

Example

C++

```
// moneypunct_pos_format.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
```

```
#include <sstream>

using namespace std;

int main() {
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true> >(loc);
   cout << loc.name( ) << " international positive number format: "
        << mpunct.pos_format( ).field[0]
        << mpunct.pos_format( ).field[1]
        << mpunct.pos_format( ).field[2]
        << mpunct.pos_format( ).field[3] << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic positive number format: "
        << mpunct2.pos_format( ).field[0]
        << mpunct2.pos_format( ).field[1]
        << mpunct2.pos_format( ).field[2]
        << mpunct2.pos_format( ).field[3] << endl;
}
```

moneypunct::positive_sign

Returns a locale-specific sequence of elements to use as a positive sign symbol.

C++

```
string_type positive_sign() const;
```

Return Value

A locale-specific sequence of elements to use as a positive sign symbol.

Remarks

The member function returns do_positive_sign.

Example

C++

```
// moneypunct_pos_sign.cpp
// compile with: /EHsc
```

```
#include <locale>
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
      use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international positive sign:"
        << mpunct.positive_sign( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic positive sign:"
        << mpunct2.positive_sign( ) << endl;

   locale loc2( "French" );

   const moneypunct <char, true> &mpunct3 =
      use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international positive sign:"
        << mpunct3.positive_sign( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic positive sign:"
        << mpunct4.positive_sign( ) << endl;
};
```

Output

```
German_Germany.1252 international positive sign:
German_Germany.1252 domestic positive sign:
French_France.1252 international positive sign:
French_France.1252 domestic positive sign:
```

moneypunct::string_type

A type that describes a string containing characters of type **CharType**.

C++

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of class template basic_string whose objects can store copies of the punctuation sequences.

moneypunct::thousands_sep

Returns a locale-specific sequence of elements to use as a thousands separator symbol.

C++

```
CharType thousands_sep() const;
```

Return Value

A locale-specific sequence of elements to use as a thousands separator

Remarks

The member function returns do_thousands_sep.

Example

C++

```
// moneypunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const moneypunct <char, true> &mpunct =
       use_facet <moneypunct <char, true > >(loc);
   cout << loc.name( ) << " international thousands separator: "
        << mpunct.thousands_sep( ) << endl;

   const moneypunct <char, false> &mpunct2 =
      use_facet <moneypunct <char, false> >(loc);
   cout << loc.name( ) << " domestic thousands separator: "
        << mpunct2.thousands_sep( ) << endl << endl;

   locale loc2( "english_canada" );
```

```
   const moneypunct <char, true> &mpunct3 =
       use_facet <moneypunct <char, true> >(loc2);
   cout << loc2.name( ) << " international thousands separator: "
        << mpunct3.thousands_sep( ) << endl;

   const moneypunct <char, false> &mpunct4 =
      use_facet <moneypunct <char, false> >(loc2);
   cout << loc2.name( ) << " domestic thousands separator: "
        << mpunct4.thousands_sep( ) << endl;
}
```

Output

```
German_Germany.1252 international thousands separator: .
German_Germany.1252 domestic thousands separator: .

English_Canada.1252 international thousands separator: ,
English_Canada.1252 domestic thousands separator: ,
```

See also

<locale>

Thread Safety in the C++ Standard Library

moneypunct_byname Class

Article • 08/03/2021

A derived class template that describes an object that can serve as a `moneypunct` facet of a given locale, enabling the formatting monetary input field or monetary output fields.

Syntax

C++

```
template <class CharType, bool Intl = false>
class moneypunct_byname : public moneypunct<CharType, Intl>
{
public:
    explicit moneypunct_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit moneypunct_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~moneypunct_byname();

};
```

Remarks

Its behavior is determined by the named locale `_Locname`. Each constructor initializes its base object with moneypunct<CharType, Intl>(`_Refs`).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

num_get Class

Article • 08/03/2021

A class template that describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to numeric values.

Syntax

C++

```
template <class CharType, class InputIterator =
istreambuf_iterator<CharType>>
class num_get : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

InputIterator
The type of iterator from which the numeric get functions read their input.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Constructors

Constructor	Description
num_get	The constructor for objects of type `num_get` that are used to extract numerical values from sequences.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.

Type name	Description
iter_type	A type that describes an input iterator.

Member functions

Member function	Description
do_get	A virtual function that is called to extracts a numerical or Boolean value from a character sequence.
get	Extracts a numerical or Boolean value from a character sequence.

Requirements

Header: <locale>

Namespace: std

num_get::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

num_get::do_get

A virtual function that is called to extracts a numerical or Boolean value from a character sequence.

```
C++

virtual iter_type do_get(
    iter_type first,
    iter_type last,
```

```
    ios_base& iosbase,
    ios_base::iostate& state,
    long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned short& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned int& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long long& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    float& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    double& val) const;

virtual iter_type do_get(
    iter_type first,
```

```
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long double& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    void *& val) const;

virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    bool& val) const;
```

Parameters

first
The beginning of the range of characters from which to read the number.

last
The end of the range of characters from which to read the number.

iosbase
The ios_base whose flags are used by the conversion.

state
The state to which failbit (see ios_base::iostate) is added upon failure.

val
The value that was read.

Return Value

The iterator after the value has been read.

Remarks

The first virtual protected member function,

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long& val) const;
```

matches sequential elements beginning at *first* in the sequence `[first, last)` until it has recognized a complete, nonempty integer input field. If successful, it converts this field to its equivalent value as type **`long`**, and stores the result in *val*. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in *val* and sets `ios_base::failbit` in `state`. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `state`.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of **`char`** elements from a file. (Each such **`char`** element is assumed to map to an equivalent element of type `Elem` by a simple, one-to-one, mapping.) The equivalent scan conversion specification is determined as follows:

If `iosbase.`ios_base::flags`() & ios_base::basefield == ios_base::`oct, the conversion specification is `lo`.

If `iosbase.flags() & ios_base::basefield == ios_base::`hex, the conversion specification is `lx`.

If `iosbase.flags() & ios_base::basefield == 0`, the conversion specification is `li`.

Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the locale facet`fac` returned by the call use_facet `<`numpunct`<Elem>(iosbase.`ios_base::getloc`())`. Specifically:

`fac.`numpunct::grouping`()` determines how digits are grouped to the left of any decimal point

`fac.`numpunct::thousands_sep`()` determines the sequence that separates groups of digits to the left of any decimal point.

If no instances of `fac.thousands_sep()` occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` are enforced and separators are removed before the scan conversion occurs.

The fourth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful it converts the numeric input field to a value of type **`unsigned long`** and stores that value in *val*.

The fifth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lld`. If successful it converts the numeric input field to a value of type **`long long`** and stores that value in *val*.

The sixth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `llu`. If successful it converts the numeric input field to a value of type **`unsigned long long`** and stores that value in *val*.

The seventh virtual protected member function:

C++

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    float& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty floating-point input field. `fac.`numpunct::decimal_point`()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The eighth virtual protected member function:

C++

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    double& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty floating-point input field. `fac.`numpunct::decimal_point`()` determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The ninth virtual protected member function:

C++

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long double& val) const;
```

behaves the same as the eighth, except that the equivalent scan conversion specifier is `Lf`.

The tenth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The last (eleventh) virtual protected member function:

```
C++
```

```
virtual iter_type do_get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    bool& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty Boolean input field. If successful it converts the Boolean input field to a value of type **`bool`** and stores that value in *val*.

A Boolean input field takes one of two forms. If `iosbase.flags() & ios_base::`boolalpha is false, it is the same as an integer input field, except that the converted value must be either 0 (for false) or 1 (for true). Otherwise, the sequence must match either `fac.`numpunct::falsename`()` (for false), or `fac.`numpunct::truename`()` (for true).

Example

See the example for get, where the virtual member function is called by `do_get`.

num_get::get

Extracts a numerical or Boolean value from a character sequence.

```
C++
```

```
iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
```

```
    ios_base::iostate& state,
    bool& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned short& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned int& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long long& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long long& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    float& val) const;

iter_type get(
    iter_type first,
    iter_type last,
```

```
    ios_base& iosbase,
    ios_base::iostate& state,
    double& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long double& val) const;

iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    void *& val) const;
```

Parameters

first
The beginning of the range of characters from which to read the number.

last
The end of the range of characters from which to read the number.

iosbase
The ios_base whose flags are used by the conversion.

state
The state to which failbit (see ios_base::iostate) is added upon failure.

val
The value that was read.

Return Value

The iterator after the value has been read.

Remarks

All member functions return do_get`( first, last, iosbase, state, val)`.

The first virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty integer input field. If successful, it converts this field to its equivalent value as

type `long` and stores the result in *val*. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in *val* and sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of `char` elements from a file. Each such `char` element is assumed to map to an equivalent element of type `CharType` by a simple, one-to-one mapping. The equivalent scan conversion specification is determined as follows:

- If `iosbase.`flags`& ios_base::basefield == ios_base::`oct, the conversion specification is `lo`.
- If `iosbase.flags & ios_base::basefield == ios_base::`hex, the conversion specification is `lx`.
- If `iosbase.flags & ios_base::basefield == 0`, the conversion specification is `li`.
- Otherwise, the conversion specification is `ld`.

The format of an integer input field is further determined by the locale facet `fac` returned by the call use_facet`<`numpunct`<Elem>(iosbase.`getloc`())`. Specifically:

- `fac.`grouping determines how digits are grouped to the left of any decimal point.
- `fac.`thousands_sep determines the sequence that separates groups of digits to the left of any decimal point.

If no instances of `fac.thousands_sep` occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping` is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

C++

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`. If successful, it converts the numeric input field to a value of type **`unsigned long`** and stores that value in *val*.

The third virtual protected member function:

```
C++
```

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    double& val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty floating-point input field. `fac.`decimal_point determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is `lf`.

The fourth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    long double& val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is `Lf`.

The fifth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is `p`.

The sixth virtual protected member function:

```
C++
```

```
virtual iter_type do_get(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    bool& val) const;
```

behaves the same as the first, except that it tries to match a complete, nonempty boolean input field. If successful it converts the Boolean input field to a value of type `bool` and stores that value in *val*.

A boolean input field takes one of two forms. If `iosbase.flags & ios_base::`boolalpha is `false`, it is the same as an integer input field, except that the converted value must be either 0 (for `false`) or 1 (for `true`). Otherwise, the sequence must match either `fac.`falsename (for `false`), or `fac.`truename (for `true`).

Example

C++

```
// num_get_get.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   basic_stringstream<char> psz, psz2;
   psz << "-1000,56";

   ios_base::iostate st = 0;
   long double fVal;
   cout << use_facet <numpunct <char> >(loc).thousands_sep( ) << endl;

   psz.imbue( loc );
   use_facet <num_get <char> >
   (loc).get( basic_istream<char>::_Iter( psz.rdbuf( ) ),
           basic_istream<char>::_Iter(0), psz, st, fVal );

   if ( st & ios_base::failbit )
      cout << "money_get( ) FAILED" << endl;
   else
      cout << "money_get( ) = " << fVal << endl;
}
```

num_get::iter_type

A type that describes an input iterator.

```
C++
```

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `InputIterator`.

num_get::num_get

The constructor for objects of type `num_get` that are used to extract numerical values from sequences.

```
C++
```

```
explicit num_get(size_t refs = 0);
```

Parameters

refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with `locale::`facet`(refs)`.

See also

<locale>

facet Class

Thread Safety in the C++ Standard Library

num_put Class

Article • 08/03/2021

A class template that describes an object that can serve as a locale facet to control conversions of numeric values to sequences of type `CharType`.

Syntax

C++

```
template <class CharType,
    class OutputIterator = ostreambuf_iterator<CharType>>
class num_put : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

OutputIterator
The type of iterator to which the numeric put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

Constructor	Description
num_put	The constructor for objects of type `num_put`.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.

Member functions

Member function	Description
do_put	A virtual function that is called to convert a number into a sequence of `CharType`s that represents the number formatted for a given locale.
put	Converts a number into a sequence of `CharType`s which represents the number formatted for a given locale.

Requirements

Header: <locale>

Namespace: std

num_put::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

num_put::do_put

A virtual function that is called to convert a number into a sequence of `CharType`s that represents the number formatted for a given locale.

```
C++

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    bool val) const;

virtual iter_type do_put(
```

```
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    unsigned long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    double val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long double val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const void* val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const long long val) const;

virtual iter_type do_put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const unsigned long long val) const;
```

Parameters

next

An iterator addressing the first element of the inserted string.

_Iosbase

Specified the stream which contains locale with the numpunct facet used to punctuate the output and flags for formatting the output.

_Fill
A character that is used for spacing.

val
The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

The first virtual protected member function generates sequential elements beginning at *next* to produce an integer output field from the value of *val*. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions for generating a series of `char` elements to a file. Each such char element is assumed to map to an equivalent element of type `CharType` by a simple, one-to-one mapping. Where a print function pads a field with either spaces or the digit 0, however, `do_put` instead uses `fill`. The equivalent print conversion specification is determined as follows:

- If **iosbase**. flags & `ios_base::basefield` == `ios_base::`oct, the conversion specification is `lo`.
- If **iosbase.flags** & **ios_base::basefield** == `ios_base::`hex, the conversion specification is `lx`.
- Otherwise, the conversion specification is `ld`.

If **iosbase**. width is nonzero, a field width of this value is prepended. The function then calls **iosbase**. **width**(0) to reset the field width to zero.

Padding occurs only if the minimum number of elements *N* required to specify the output field is less than **iosbase**. width. Such padding consists of a sequence of *N* - **width** copies of **fill**. Padding then occurs as follows:

- If **iosbase**. **flags** & `ios_base::adjustfield` == `ios_base::`left, the flag **-** is prepended. (Padding occurs after the generated text.)

- If **iosbase.flags** & **ios_base::adjustfield** == `ios_base::`internal, the flag **0** is prepended. (For a numeric output field, padding occurs where the print functions pad with 0.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:

- If **iosbase**. **flags** & `ios_base::`showpos is nonzero, the flag + is prepended to the conversion specification.
- If **iosbase**. **flags** & **ios_base::**showbase is nonzero, the flag # is prepended to the conversion specification.

The format of an integer output field is further determined by the locale facet**fac** returned by the call use_facet < numpunct< **Elem**>(**iosbase**. getloc). Specifically:

- **fac**. grouping determines how digits are grouped to the left of any decimal point
- **fac**. thousands_sep determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by **fac**. **grouping** (its first element has the value CHAR_MAX), then no instances of **fac**. `thousands_sep` are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
C++

virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    unsigned long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lu`.

The third virtual protected member function:

```
C++

virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    double val) const;
```

behaves the same as the first, except that it produces a floating-point output field from the value of **val**. **fac**. decimal_point determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specification is determined as follows:

- If **iosbase**. **flags** & `ios_base::floatfield` == `ios_base::`fixed, the conversion specification is `lf`.
- If **iosbase**. **flags** & **ios_base::floatfield** == `ios_base::`scientific, the conversion specification is `le`. If **iosbase**. **flags** & `ios_base::`uppercase is nonzero, `e` is replaced with `E`.
- Otherwise, the conversion specification is **lg**. If **iosbase**. **flags** & **ios_base::uppercase** is nonzero, `g` is replaced with `G`.

If **iosbase**. **flags** & **ios_base::fixed** is nonzero or if **iosbase**. precision is greater than zero, a precision with the value **iosbase**. **precision** is prepended to the conversion specification. Any padding behaves the same as for an integer output field. The padding character is **fill**. Finally:

- If **iosbase**. **flags** & `ios_base::`showpos is nonzero, the flag + is prepended to the conversion specification.
- If **iosbase**. **flags** & `ios_base::`showpoint is nonzero, the flag # is prepended to the conversion specification.

The fourth virtual protected member function:

```
C++
```

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    long double val) const;
```

behaves the same the third, except that the qualifier `l` in the conversion specification is replaced with `L`.

The fifth virtual protected member function:

```
C++
```

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
```

```
    CharType _Fill,
    const void* val) const;
```

behaves the same the first, except that the conversion specification is `p`, plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
C++
```

```
virtual iter_type do_put(iter_type next,
    ios_base& _Iosbase,
    CharType _Fill,
    bool val) const;
```

behaves the same as the first, except that it generates a Boolean output field from *val*.

A Boolean output field takes one of two forms. If `iosbase.flags & ios_base::`boolalpha is **`false`**, the member function returns `do_put(_Next, _Iosbase, _Fill, (long)val)`, which typically produces a generated sequence of either 0 (for **`false`**) or 1 (for **`true`**). Otherwise, the generated sequence is either *fac*.falsename (for **`false`**), or *fac*.truename (for **`true`**).

The seventh virtual protected member function:

```
C++
```

```
virtual iter_type do_put(iter_type next,
    ios_base& iosbase,
    Elem fill,
    long long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `lld`.

The eighth virtual protected member function:

```
C++
```

```
virtual iter_type do_put(iter_type next,
    ios_base& iosbase,
    Elem fill,
    unsigned long long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of `ld` with `llu`.

Example

See the example for put, which calls `do_put`.

num_put::iter_type

A type that describes an output iterator.

```
C++

typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **OutputIterator.**

num_put::num_put

The constructor for objects of type `num_put`.

```
C++

explicit num_put(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.

- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(_ *Refs*).

num_put::put

Converts a number into a sequence of `CharType`s that represents the number formatted for a given locale.

C++

```
iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    bool val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    unsigned long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    Long long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    Unsigned long long val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    double val) const;

iter_type put(
    iter_type dest,
```

```
    ios_base& _Iosbase,
    _Elem _Fill,
    long double val) const;

iter_type put(
    iter_type dest,
    ios_base& _Iosbase,
    _Elem _Fill,
    const void* val) const;
```

Parameters

dest
An iterator addressing the first element of the inserted string.

_Iosbase
Specified the stream that contains locale with the numpunct facet used to punctuate the output and flags for formatting the output.

_Fill
A character that is used for spacing.

val
The number or Boolean type that is to be output.

Return Value

An output iterator the addresses the position one beyond the last element produced.

Remarks

All member functions return do_put(`next`, `_Iosbase`, `_Fill`, `val`).

Example

C++

```
// num_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
```

```
    locale loc( "german_germany" );
    basic_stringstream<char> psz2;
    ios_base::iostate st = 0;
    long double fVal;
    cout << "The thousands separator is: "
         << use_facet < numpunct <char> >(loc).thousands_sep( )
         << endl;

    psz2.imbue( loc );
    use_facet < num_put < char > >
       ( loc ).put(basic_ostream<char>::_Iter(psz2.rdbuf( ) ),
                   psz2, ' ', fVal=1000.67);

    if ( st & ios_base::failbit )
       cout << "num_put( ) FAILED" << endl;
    else
       cout << "num_put( ) = " << psz2.rdbuf( )->str( ) << endl;
}
```

Output

```
The thousands separator is: .
num_put( ) = 1.000,67
```

See also

<locale>

facet Class

Thread Safety in the C++ Standard Library

numpunct Class

Article • 08/03/2021

A class template that describes an object that can serve as a local facet to describe the sequences of type `CharType` used to represent information about the formatting and punctuation of numeric and Boolean expressions.

Syntax

C++

```
template <class CharType>
class numpunct : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters in a locale.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id.**

Constructors

Constructor	Description
numpunct	The constructor for objects of type `numpunct`.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.
string_type	A type that describes a string containing characters of type `CharType`.

Member functions

Member function	Description
decimal_point	Returns a locale-specific element to use as a decimal point.
do_decimal_point	A protected virtual member function that is called to return a locale-specific element to use as a decimal point.
do_falsename	A protected virtual member function that is called to return a string to use as a text representation of the value `false`.
do_grouping	A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.
do_thousands_sep	A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.
do_truename	A protected virtual member function that is called to return a string to use as a text representation of the value `true`.
falsename	Returns a string to use as a text representation of the value `false`.
grouping	Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.
thousands_sep	Returns a locale-specific element to use as a thousands separator.
truename	Returns a string to use as a text representation of the value `true`.

Requirements

Header: <locale>

Namespace: std

numpunct::char_type

A type that is used to describe a character used by a locale.

```
C++

typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

numpunct::decimal_point

Returns a locale-specific element to use as a decimal point.

```
C++
```

```
CharType decimal_point() const;
```

Return Value

A locale-specific element to use as a decimal point.

Remarks

The member function returns do_decimal_point.

Example

```
C++
```

```
// numpunct_decimal_point.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const numpunct <char> &npunct =
   use_facet <numpunct <char> >( loc);
   cout << loc.name( ) << " decimal point "<<
   npunct.decimal_point( ) << endl;
   cout << loc.name( ) << " thousands separator "
   << npunct.thousands_sep( ) << endl;
};
```

```
Output
```

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

numpunct::do_decimal_point

A protected virtual member function that is called to return a locale-specific element to use as a decimal point.

```
C++
```

```
virtual CharType do_decimal_point() const;
```

Return Value

A locale-specific element to use as a decimal point.

Example

See the example for decimal_point, where the virtual member function is called by `decimal_point`.

numpunct::do_falsename

The protected virtual member function returns a sequence to use as a text representation of the value **`false`**.

```
C++
```

```
virtual string_type do_falsename() const;
```

Return Value

A string containing a sequence to use as a text representation of the value **`false`**.

Remarks

The member function returns the string "false" to represent the value **`false`** in all locales.

Example

See the example for falsename, where the virtual member function is called by `falsename`.

numpunct::do_grouping

A protected virtual member function that is called to return a locale-specific rule for determining how digits are grouped to the left of any decimal point.

```
C++
```

```
virtual string do_grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for **Iconv::grouping**.

Example

See the example for grouping, where the virtual member function is called by `grouping`.

numpunct::do_thousands_sep

A protected virtual member function that is called to return a locale-specific element to use as a thousands separator.

```
C++
```

```
virtual CharType do_thousands_sep() const;
```

Return Value

Returns a locale-specific element to use as a thousands separator.

Remarks

The protected virtual member function returns a locale-specific element of type `CharType` to use as a group separator to the left of any decimal point.

Example

See the example for thousands_sep, where the virtual member function is called by `thousands_sep`.

numpunct::do_truename

A protected virtual member function that is called to return a string to use as a text representation of the value **`true`**.

```
C++
```

```
virtual string_type do_truename() const;
```

Remarks

A string to use as a text representation of the value **`true`**.

All locales return a string "true" to represent the value **`true`**.

Example

See the example for truename, where the virtual member function is called by `truename`.

numpunct::falsename

Returns a string to use as a text representation of the value **`false`**.

```
C++
```

```
string_type falsename() const;
```

Return Value

A string containing a sequence of `CharType`s to use as a text representation of the value `false`.

Remarks

The member function returns the string "false" to represent the value `false` in all locales.

The member function returns do_falsename.

Example

C++

```
// numpunct_falsename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "English" );

   const numpunct <char> &npunct = use_facet <numpunct <char> >( loc );
   cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
   cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

   locale loc2( "French" );
   const numpunct <char> &npunct2 = use_facet <numpunct <char> >(loc2);
   cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
   cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}
```

Output

```
English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false
```

numpunct::grouping

Returns a locale-specific rule for determining how digits are grouped to the left of any decimal point.

C++

```
string grouping() const;
```

Return Value

A locale-specific rule for determining how digits are grouped to the left of any decimal point.

Remarks

The member function returns do_grouping.

Example

C++

```
// numpunct_grouping.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany");

   const numpunct <char> &npunct =
       use_facet < numpunct <char> >( loc );
   for (unsigned int i = 0; i < npunct.grouping( ).length( ); i++)
   {
      cout << loc.name( ) << " international grouping:\n the "
           << i <<"th group to the left of the radix character "
           << "is of size " << (int)(npunct.grouping ( )[i])
           << endl;
   }
}
```

Output

```
German_Germany.1252 international grouping:
the 0th group to the left of the radix character is of size 3
```

numpunct::numpunct

The constructor for objects of type `numpunct`.

```
C++
```

```
explicit numpunct(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale**::facet(`_Refs`).

numpunct::string_type

A type that describes a string containing characters of type **CharType**.

```
C++
```

```
typedef basic_string<CharType, Traits, Allocator> string_type;
```

Remarks

The type describes a specialization of class template basic_string whose objects can store copies of the punctuation sequences.

numpunct::thousands_sep

Returns a locale-specific element to use as a thousands separator.

C++

```
CharType thousands_sep() const;
```

Return Value

A locale-specific element to use as a thousands separator.

Remarks

The member function returns do_thousands_sep.

Example

C++

```
// numpunct_thou_sep.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "german_germany" );

   const numpunct <char> &npunct =
   use_facet < numpunct < char > >( loc );
   cout << loc.name( ) << " decimal point "<<
   npunct.decimal_point( ) << endl;
   cout << loc.name( ) << " thousands separator "
   << npunct.thousands_sep( ) << endl;
};
```

Output

```
German_Germany.1252 decimal point ,
German_Germany.1252 thousands separator .
```

numpunct::truename

Returns a string to use as a text representation of the value `true`.

C++

```
string_type falsename() const;
```

Return Value

A string to use as a text representation of the value `true`.

Remarks

The member function returns do_truename.

All locales return a string "true" to represent the value `true`.

Example

C++

```
// numpunct_truename.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
{
   locale loc( "English" );

   const numpunct < char> &npunct = use_facet <numpunct <char> >( loc );
   cout << loc.name( ) << " truename "<< npunct.truename( ) << endl;
   cout << loc.name( ) << " falsename "<< npunct.falsename( ) << endl;

   locale loc2("French");
   const numpunct <char> &npunct2 = use_facet <numpunct <char> >( loc2 );
   cout << loc2.name( ) << " truename "<< npunct2.truename( ) << endl;
   cout << loc2.name( ) << " falsename "<< npunct2.falsename( ) << endl;
}
```

Output

```
English_United States.1252 truename true
English_United States.1252 falsename false
French_France.1252 truename true
French_France.1252 falsename false
```

See also

<locale>

facet Class

Thread Safety in the C++ Standard Library

numpunct_byname Class

Article • 08/03/2021

The derived class template describes an object that can serve as a `numpunct` facet of a given locale enabling the formatting and punctuation of numeric and Boolean expressions.

Syntax

C++

```
template <class CharType>
class numpunct_byname : public numpunct<Elem> {
public:
    explicit numpunct_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit numpunct_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~numpunct_byname();

};
```

Remarks

Its behavior is determined by the named locale `_Locname`. The constructor initializes its base object with numpunct<CharType>(`_Refs`).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

time_base Class

Article • 08/03/2021

The class serves as a base class for facets of class template time_get, defining just the enumerated type `dateorder` and several constants of this type.

Syntax

C++

```
class time_base : public locale::facet {
public:
    enum dateorder {
        no_order,
        dmy,
        mdy,
        ymd,
        ydm
    };
    time_base(size_t _Refs = 0)
    ~time_base();
};
```

Remarks

Each constant characterizes a different way to order the components of a date. The constants are:

- `no_order` specifies no particular order.
- `dmy` specifies the order day, month, then year, as in 2 December 1979.
- `mdy` specifies the order month, day, then year, as in December 2, 1979.
- `ymd` specifies the order year, month, then day, as in 1979/12/2.
- `ydm` specifies the order year, day, then month, as in 1979: 2 Dec.

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

time_get Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to control conversions of sequences of type `CharType` to time values.

Syntax

```
C++
```

```
template <class CharType,
    class InputIterator = istreambuf_iterator<CharType>>
class time_get : public time_base;
```

Parameters

CharType
The type used within a program to encode characters.

InputIterator
The iterator from which the time values are read.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

Constructor	Description
time_get	The constructor for objects of type `time_get`.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an input iterator.

Member functions

Member function	Description
date_order	Returns the date order used by a facet.
do_date_order	A protected virtual member function that is called to return the date order used by a facet.
do_get	Reads and converts character data to a time value.
do_get_date	A protected virtual member function that is called to parse a string as the date produced by the `x` specifier for `strftime`.
do_get_monthname	A protected virtual member function that is called to parse a string as the name of the month.
do_get_time	A protected virtual member function that is called to parse a string as the date produced by the `X` specifier for `strftime`.
do_get_weekday	A protected virtual member function that is called to parse a string as the name of the day of the week.
do_get_year	A protected virtual member function that is called to parses a string as the name of the year.
get	Reads from a source of character data and converts that data to a time that is stored in a time struct.
get_date	Parses a string as the date produced by the `x` specifier for `strftime`.
get_monthname	Parses a string as the name of the month.
get_time	Parses a string as the date produced by the `X` specifier for `strftime`.
get_weekday	Parses a string as the name of the day of the week.
get_year	Parses a string as the name of the year.

Requirements

Header: <locale>

Namespace: std

time_get::char_type

A type that is used to describe a character used by a locale.

```
C++
```

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter **CharType**.

time_get::date_order

Returns the date order used by a facet.

```
C++
```

```
dateorder date_order() const;
```

Return Value

The date order used by a facet.

Remarks

The member function returns do_date_order.

Example

```
C++
```

```
// time_get_date_order.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
void po( char *p )
{
   locale loc( p );

   time_get <char>::dateorder order = use_facet <time_get <char> >( loc
).date_order ( );
   cout << loc.name( );
   switch (order){
      case time_base::dmy: cout << "(day, month, year)" << endl;
```

```
        break;
        case time_base::mdy: cout << "(month, day, year)" << endl;
        break;
        case time_base::ydm: cout << "(year, day, month)" << endl;
        break;
        case time_base::ymd: cout << "(year, month, day)"<< endl;
        break;
        case time_base::no_order: cout << "(no_order)"<< endl;
        break;
    }
}

int main( )
{
    po( "C" );
    po( "german" );
    po( "English_Britain" );
}
```

Output

```
C(month, day, year)
German_Germany.1252(day, month, year)
English_United Kingdom.1252(day, month, year)
```

time_get::do_date_order

A protected virtual member function that is called to return the date order used by a facet.

C++

```
virtual dateorder do_date_order() const;
```

Return Value

The date order used by a facet.

Remarks

The virtual protected member function returns a value of type **time_base::dateorder**, which describes the order in which date components are matched by do_get_date. In this implementation, the value is **time_base::mdy**, corresponding to dates of the form December 2, 1979.

Example

See the example for date_order, which calls `do_date_order`.

time_get::do_get

Reads and converts character data to a time value. Accepts one conversion specifier and modifier.

```
C++
```

```
virtual iter_type
    do_get(
iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm,
    char fmt,
    char mod) const;
```

Parameters

first
An Input iterator that indicates the start of the sequence to convert.

last
An Input iterator that indicates the end of the sequence.

iosbase
A stream object.

state
A field in iosbase where appropriate bitmask elements are set to indicate errors.

ptm
A pointer to the time structure where the time is to be stored.

fmt
A conversion specifier character.

mod
An optional modifier character.

Return Value

Returns an iterator that designates the first unconverted element. A conversion failure sets `ios_base::failbit` in `state` and returns *first*.

Remarks

The virtual member function converts and skips one or more input elements in the range [`first`, `last`) to determine the values stored in one or more members of `*pt`. A conversion failure sets `ios_base::failbit` in `state` and returns *first*. Otherwise, the function returns an iterator designating the first unconverted element.

The conversion specifiers are:

`'a'` or `'A'` -- behaves the same as time_get::get_weekday.

`'b'`, `'B'`, or `'h'` -- behaves the same as time_get::get_monthname.

`'c'` -- behaves the same as `"%b %d %H : %M : %S %Y"`.

`'C'` -- converts a decimal input field in the range [0, 99] to the value `val` and stores `val * 100 - 1900` in `pt-&tm_year`.

`'d'` or `'e'` -- converts a decimal input field in the range [1, 31] and stores its value in `pt-&tm_mday`.

`'D'` -- behaves the same as `"%m / %d / %y"`.

`'H'` -- converts a decimal input field in the range [0, 23] and stores its value in `pt-&tm_hour`.

`'I'` -- converts a decimal input field in the range [0, 11] and stores its value in `pt-&tm_hour`.

`'j'` -- converts a decimal input field in the range [1, 366] and stores its value in `pt-&tm_yday`.

`'m'` -- converts a decimal input field in the range [1, 12] to the value `val` and stores `val - 1` in and stores its value in `pt-&tm_mon`.

`'M'` -- converts a decimal input field in the range [0, 59] and stores its value in `pt-&tm_min`.

`'n'` or `'t'` -- behaves the same as `" "`.

`'p'` -- converts "AM" or "am" to zero and "PM" or "PM" to 12 and adds this value to `pt-&tm_hour`.

`'r'` -- behaves the same as `"%I : %M : %S %p"`.

`'R'` -- behaves the same as `"%H %M"`.

`'S'` -- converts a decimal input field in the range [0, 59] and stores its value in `pt-&tm_sec`.

`'T'` or `'X'` -- behaves the same as `"%H : %M : S"`.

`'U'` -- converts a decimal input field in the range [0, 53] and stores its value in `pt-&tm_yday`.

`'w'` -- converts a decimal input field in the range [0, 6] and stores its value in `pt-&tm_wday`.

`'W'` -- converts a decimal input field in the range [0, 53] and stores its value in `pt-&tm_yday`.

`'x'` -- behaves the same as `"%d / %m / %y"`.

`'y'` -- converts a decimal input field in the range [0, 99] to the value `val` and stores `val < 69 val + 100 : val` in `pt-&tm_year`.

`'Y'` -- behaves the same as time_get::get_year.

Any other conversion specifier sets `ios_base::failbit` in `state` and returns. In this implementation, any modifier has no effect.

time_get::do_get_date

A protected virtual member function that is called to parse a string as the date produced by the *x* specifier for `strftime`.

C++

```
virtual iter_type do_get_date(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty date input field. If successful, it converts this field to its equivalent value as the components **tm::tm_mon**, **tm::tm_day**, and **tm::tm_year**, and stores the results in `ptm->tm_mon`, `ptm->tm_day`, and `ptm->tm_year`, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets `iosbase::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The format for the date input field is locale dependent. For the default locale, the date input field has the form MMM DD, YYYY, where:

- MMM is matched by calling get_monthname, giving the month.
- DD is a sequence of decimal digits whose corresponding numeric value must be in the range [1, 31], giving the day of the month.

- YYYY is matched by calling get_year, giving the year.

The literal spaces and commas must match corresponding elements in the input sequence.

Example

See the example for get_date, which calls `do_get_date`.

time_get::do_get_monthname

A protected virtual member function that is called to parse a string as the name of the month.

```
C++
```

```
virtual iter_type do_get_monthname(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
Unused.

state
An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty month input field. If successful, it converts this field to its equivalent value as the component **tm::tm_mon**, and stores the result in `ptm->tm_mon`. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as Jan, January, Feb, February, and so on. The converted value is the number of months since January.

Example

See the example for get_monthname, which calls `do_get_monthname`.

time_get::do_get_time

A protected virtual member function that is called to parse a string as the date produced by the *X* specifier for `strftime`.

C++

```
virtual iter_type do_get_time(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
Unused.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at first in the sequence [`first`, `last`) until it has recognized a complete, nonempty time input field. If successful, it converts this field to its equivalent value as the components `tm::tm_hour`, `tm::tm_min`, and `tm::tm_sec`, and stores the results in `ptm->tm_hour`, `ptm->tm_min`, and `ptm->tm_sec`, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

In this implementation, the time input field has the form HH:MM:SS, where:

- HH is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 24), giving the hour of the day.
- MM is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the minutes past the hour.
- SS is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the seconds past the minute.

The literal colons must match corresponding elements in the input sequence.

Example

See the example for get_time, which calls `do_get_time`.

time_get::do_get_weekday

A protected virtual member function that is called to parse a string as the name of the day of the week.

```
C++
```

```
virtual iter_type do_get_weekday(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *first* in the sequence [`first`, `last`) until it has recognized a complete, nonempty weekday input field. If successful, it converts this field to its equivalent value as the component **tm::tm_wday**, and stores the result in `ptm->tm_wday`. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element

beyond any prefix of a valid weekday input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as Sun, Sunday, Mon, Monday, and so on. The converted value is the number of days since Sunday.

Example

See the example for get_weekday, which calls `do_get_weekday`.

time_get::do_get_year

A protected virtual member function that is called to parses a string as the name of the year.

```
C++
```

```
virtual iter_type do_get_year(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The virtual protected member function tries to match sequential elements beginning at *first* in the sequence [`first`, `last`) until it has recognized a complete, nonempty year input field. If successful, it converts this field to its equivalent value as the component **tm::tm_year**, and stores the result in `ptm->tm_year`. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets `ios_base::failbit` in *state*. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals *last*, the function sets `ios_base::eofbit` in *state*.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range [1900, 2036). The stored value is this value minus 1900. In this implementation, values in the range [69, 136) represent the range of years [1969, 2036). Values in the range [0, 69) are also permissible, but may represent either the range of years [1900, 1969) or [2000, 2069), depending on the specific translation environment.

Example

See the example for get_year, which calls `do_get_year`.

time_get::get

Reads from a source of character data and converts that data to a time that is stored in a time struct. The first function accepts one conversion specifier and modifier, the second accepts several.

C++

```
iter_type get(
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm,
    char fmt,
    char mod) const;

iter_type get(
```

```
    iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm,
    char_type* fmt_first,
    char_type* fmt_last) const;
```

Parameters

first
Input iterator that indicates where the sequence to be converted starts.

last
Input iterator that indicates the end of the sequence to be converted.

iosbase
The stream.

state
The appropriate bitmask elements are set for the stream state to indicate errors.

ptm
Pointer to the time structure where the time is to be stored.

fmt
A conversion specifier character.

mod
An optional modifier character.

fmt_first
Points to where the format directives start.

fmt_last
Points to the end of the format directives.

Return Value

Returns an iterator to the first character after the data that was used to assign the time struct `*ptm`.

Remarks

The first member function returns `do_get(first, last, iosbase, state, ptm, fmt, mod)`.

The second member function calls `do_get` under the control of the format delimited by `[fmt_first, fmt_last)`. It treats the format as a sequence of fields, each of which determines the conversion of zero or more input elements delimited by `[first, last)`. It returns an iterator designating the first unconverted element. There are three kinds of fields:

A per cent (%) in the format, followed by an optional modifier *mod* in the set [EOQ#], followed by a conversion specifier *fmt*, replaces *first* with the value returned by `do_get(first, last, iosbase, state, ptm, fmt, mod)`. A conversion failure sets `ios_base::failbit` in *state* and returns.

A whitespace element in the format skips past zero or more input whitespace elements.

Any other element in the format must match the next input element, which is skipped. A match failure sets `ios_base::failbit` in *state* and returns.

time_get::get_date

Parses a string as the date produced by the *x* specifier for `strftime`.

```
C++
```

```
iter_type get_date(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_date(`first`, `last`, `iosbase`, `state`, `ptm`).

Note that months are counted from 0 to 11.

Example

C++

```
// time_get_get_date.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset(&t, 0, sizeof(struct tm));

   pszGetF << "July 4, 2000";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet <time_get<char> >
   (loc).get_date(basic_istream<char>::_Iter(pszGetF.rdbuf( ) ),
            basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if ( st & ios_base::failbit )
      cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: "
<< *i << endl;
   else

      cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
```

```
      << "\ntm_sec: " << t.tm_sec
      << "\ntm_min: " << t.tm_min
      << "\ntm_hour: " << t.tm_hour
      << "\ntm_mday: " << t.tm_mday
      << "\ntm_mon: " << t.tm_mon
      << "\ntm_year: " << t.tm_year
      << "\ntm_wday: " << t.tm_wday
      << "\ntm_yday: " << t.tm_yday
      << "\ntm_isdst: " << t.tm_isdst
      << endl;
}
```

Output

```
time_get(July 4, 2000) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 4
tm_mon: 6
tm_year: 100
tm_wday: 0
tm_yday: 0
tm_isdst: 0
```

time_get::get_monthname

Parses a string as the name of the month.

C++

```
iter_type get_monthname(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
Unused.

state
An output parameter that sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the month information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_monthname(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

C++

```
// time_get_get_monthname.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc ( "French" );
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "juillet";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet <time_get <char> >
   (loc).get_monthname(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
              basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: "
<< *i << endl;
   else
```

```
        cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="
        << "\ntm_sec: " << t.tm_sec
        << "\ntm_min: " << t.tm_min
        << "\ntm_hour: " << t.tm_hour
        << "\ntm_mday: " << t.tm_mday
        << "\ntm_mon: " << t.tm_mon
        << "\ntm_year: " << t.tm_year
        << "\ntm_wday: " << t.tm_wday
        << "\ntm_yday: " << t.tm_yday
        << "\ntm_isdst: " << t.tm_isdst
        << endl;
}
```

Output

```
time_get(juillet) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 0
tm_mon: 6
tm_year: 0
tm_wday: 0
tm_yday: 0
tm_isdst: 0
```

time_get::get_time

Parses a string as the date produced by the *X* specifier for `strftime`.

C++

```
iter_type get_time(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
Unused.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the date information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_time(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

C++

```
// time_get_get_time.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "11:13:20";
   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet
      <time_get <char> >
      (loc).get_time(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED
on char: " << *i << endl;
   else
```

```
      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
      << "\ntm_sec: " << t.tm_sec
      << "\ntm_min: " << t.tm_min
      << "\ntm_hour: " << t.tm_hour
      << endl;
}
```

Output

```
time_get::get_time(11:13:20) =
tm_sec: 20
tm_min: 13
tm_hour: 11
```

time_get::get_weekday

Parses a string as the name of the day of the week.

C++

```
iter_type get_weekday(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the weekday information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_weekday(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

C++

```
// time_get_get_weekday.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc ( "French" );
   basic_stringstream< char > pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "mercredi";
   pszGetF.imbue(loc);
   basic_istream<char>::_Iter i = use_facet
      <time_get<char> >
      (loc).get_weekday(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") FAILED
on char: " << *i << endl;
   else

      cout << "time_get::get_time("<< pszGetF.rdbuf( )->str( )<< ") ="
      << "\ntm_wday: " << t.tm_wday
      << endl;
}
```

Output

```
time_get::get_time(mercredi) =
tm_wday: 3
```

time_get::get_year

Parses a string as the name of the year.

```
C++
```

```
iter_type get_year(iter_type first,
    iter_type last,
    ios_base& iosbase,
    ios_base::iostate& state,
    tm* ptm) const;
```

Parameters

first
Input iterator addressing the beginning of the sequence to be converted.

last
Input iterator addressing the end of the sequence to be converted.

iosbase
A format flag which when set indicates that the currency symbol is optional; otherwise, it is required.

state
Sets the appropriate bitmask elements for the stream state according to whether the operations succeeded.

ptm
A pointer to where the year information is to be stored.

Return Value

An input iterator addressing the first element beyond the input field.

Remarks

The member function returns do_get_year(`first`, `last`, `iosbase`, `state`, `ptm`).

Example

```
C++
```

```
// time_get_get_year.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszGetF, pszPutF, pszGetI, pszPutI;
   ios_base::iostate st = 0;
   struct tm t;
   memset( &t, 0, sizeof( struct tm ) );

   pszGetF << "1928";

   pszGetF.imbue( loc );
   basic_istream<char>::_Iter i = use_facet
      <time_get<char> >
      (loc).get_year(basic_istream<char>::_Iter(pszGetF.rdbuf( )),
               basic_istream<char>::_Iter(0), pszGetF, st, &t);

   if (st & ios_base::failbit)
      cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") FAILED
on char: " << *i << endl;
   else

      cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") ="
      << "\ntm_year: " << t.tm_year
      << endl;
}
```

Output

```
time_get::get_year(1928) =
tm_year: 28
```

time_get::iter_type

A type that describes an input iterator.

C++

```
typedef InputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter **InputIterator**.

time_get::time_get

The constructor for objects of type `time_get`.

```
C++
```

```
explicit time_get(size_t refs = 0);
```

Parameters

refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.
- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

No direct examples are possible, because the destructor is protected.

The constructor initializes its base object with **locale::**facet(`refs`).

See also

<locale>
time_base Class
Thread Safety in the C++ Standard Library

time_get_byname Class

Article • 08/03/2021

The derived class template describes an object that can serve as a locale facet of type `time_get`<CharType, InputIterator>.

Syntax

```
C++
```

```
template <class Elem, class InputIterator =
    istreambuf_iterator<CharType, char_traits<CharType>>>
class time_get_byname : public time_get<CharType, InputIterator>
{
public:
    explicit time_get_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit time_get_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~time_get_byname()
};
```

Parameters

_Locname
A named locale.

_Refs
An initial reference count.

Requirements

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with time_get<CharType, InputIterator>(`_Refs`).

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

time_put Class

Article • 08/03/2021

The class template describes an object that can serve as a locale facet to control conversions of time values to sequences of type `CharType`.

Syntax

C++

```
template <class CharType,
    class OutputIterator = ostreambuf_iterator<CharType>>
class time_put : public locale::facet;
```

Parameters

CharType
The type used within a program to encode characters.

OutputIterator
The type of iterator into which the time put functions write their output.

Remarks

As with any locale facet, the static object ID has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in **id**.

Constructors

Constructor	Description
time_put	The constructor for objects of type `time_put`.

Typedefs

Type name	Description
char_type	A type that is used to describe a character used by a locale.
iter_type	A type that describes an output iterator.

Member functions

Member function	Description
do_put	A virtual function that outputs time and date information as a sequence of `CharType`s.
put	Outputs time and date information as a sequence of `CharType`s.

Requirements

Header: <locale>

Namespace: std

time_put::char_type

A type that is used to describe a character used by a locale.

```
C++
```

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

time_put::do_put

A virtual function that outputs time and date information as a sequence of `CharType`s.

```
C++
```

```
virtual iter_type do_put(
    iter_type next,
    ios_base& _Iosbase,
    const tm* _Pt,
    char _Fmt,
    char _Mod = 0) const;
```

Parameters

next
An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase
Unused.

_Pt
The time and date information being output.

_Fmt
The format of the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_Mod
A modifier for the format. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The virtual protected member function generates sequential elements beginning at `next` from time values stored in the object * `_Pt`, of type `tm`. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by `strftime`, with a last argument of *_Pt*, for generating a series of **`char`** elements into an array. Each such **`char`** element is assumed to map to an equivalent element of type `CharType` by a simple, one-to-one mapping. If *_Mod* equals zero, the effective format is "%F", where F is replaced by *_Fmt*. Otherwise, the effective format is "%MF", where M is replaced by *_Mod*.

Example

See the example for put, which calls `do_put`.

time_put::iter_type

A type that describes an output iterator.

```
C++
```

```
typedef OutputIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `OutputIterator`.

time_put::put

Outputs time and date information as a sequence of `CharType`s.

```
C++
```

```
iter_type put(iter_type next,
    ios_base& _Iosbase,
    char_type _Fill,
    const tm* _Pt,
    char _Fmt,
    char _Mod = 0) const;

iter_type put(iter_type next,
    ios_base& _Iosbase,
    char_type _Fill,
    const tm* _Pt,
    const CharType* first,
    const CharType* last) const;
```

Parameters

next
An output iterator where the sequence of characters representing time and date are to be inserted.

_Iosbase
Unused.

_Fill
The character of type `CharType` used for spacing.

_Pt
The time and date information being output.

_Fmt
The format of the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

_Mod
A modifier for the format. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

first
The beginning of the formatting string for the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

last
The end of the formatting string for the output. See strftime, wcsftime, _strftime_l, _wcsftime_l for valid values.

Return Value

An iterator to the first position after the last element inserted.

Remarks

The first member function returns do_put(`next`, `_Iosbase`, `_Fill`, `_Pt`, `_Fmt`, `_Mod`). The second member function copies to * `next` ++ any element in the interval [`first`, `last`) other than a percent (%). For a percent followed by a character *C* in the interval [`first`, `last`), the function instead evaluates `next` = `do_put`(`next`, `_Iosbase`, `_Fill`, `_Pt`, *C*, 0) and skips past *C*. If, however, *C* is a qualifier character from the set EOQ#, followed by a character `C2` in the interval [`first`, `last`), the function instead evaluates `next` = `do_put`(`next`, `_Iosbase`, `_Fill`, `_Pt`, `C2`, *C*) and skips past `C2`.

Example

C++

```
// time_put_put.cpp
// compile with: /EHsc
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
int main( )
{
   locale loc;
   basic_stringstream<char> pszPutI;
   ios_base::iostate st = 0;
```

```
    struct tm t;
    memset( &t, 0, sizeof( struct tm ) );

    t.tm_hour = 5;
    t.tm_min = 30;
    t.tm_sec = 40;
    t.tm_year = 00;
    t.tm_mday = 4;
    t.tm_mon = 6;

    pszPutI.imbue( loc );
    char *pattern = "x: %X %x";
    use_facet <time_put <char> >
    (loc).put(basic_ostream<char>::_Iter(pszPutI.rdbuf( )),
           pszPutI, ' ', &t, pattern, pattern+strlen(pattern));

       cout << "num_put( ) = " << pszPutI.rdbuf( )->str( ) << endl;

       char strftimebuf[255];
       strftime(&strftimebuf[0], 255, pattern, &t);
       cout << "strftime( ) = " << &strftimebuf[0] << endl;
}
```

Output

```
num_put( ) = x: 05:30:40 07/04/00
strftime( ) = x: 05:30:40 07/04/00
```

time_put::time_put

Constructor for objects of type `time_put`.

C++

```
explicit time_put(size_t _Refs = 0);
```

Parameters

_Refs
Integer value used to specify the type of memory management for the object.

Remarks

The possible values for the *_Refs* parameter and their significance are:

- 0: The lifetime of the object is managed by the locales that contain it.

- 1: The lifetime of the object must be manually managed.
- > 1: These values are not defined.

The constructor initializes its base object with locale::facet(_*Refs*).

See also

<locale>
time_base Class
Thread Safety in the C++ Standard Library

time_put_byname Class

Article • 08/03/2021

The derived class template describes an object that can serve as a locale facet of type `time_put` < CharType, OutputIterator >.

Syntax

C++

```
template <class CharType, class OutIt = ostreambuf_iterator<CharType,
char_traits<CharType>>>
class time_put_byname : public time_put<CharType, OutputIterator>
{
public:
    explicit time_put_byname(
    const char* _Locname,
    size_t _Refs = 0);

    explicit time_put_byname(
    const string& _Locname,
    size_t _Refs = 0);

protected:
    virtual ~time_put_byname();

};
```

Parameters

_Locname
A locale name.

_Refs
An initial reference count.

Remarks

Its behavior is determined by the named locale *_Locname*. Each constructor initializes its base object with time_put<CharType, OutputIterator>(`_Refs`).

Requirements

Header: <locale>

Namespace: std

See also

Thread Safety in the C++ Standard Library

`wbuffer_convert` class

Article • 08/05/2024

Describes a stream buffer that controls the transmission of elements to and from a byte stream buffer.

Syntax

```
C++
```

```
template <class Codecvt, class Elem = wchar_t, class Traits =
std::char_traits<Elem>>
class wbuffer_convert
    : public std::basic_streambuf<Elem, Traits>
```

Parameters

`Codecvt`
The locale facet that represents the conversion object.

`Elem`
The wide-character element type.

`Traits`
The traits associated with *`Elem`*.

Remarks

This class template describes a stream buffer that controls the transmission of elements of type `_Elem`, whose character traits are described by the class `Traits`, to and from a byte stream buffer of type `std::streambuf`.

Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

An object of this class template stores:

- A pointer to its underlying byte stream buffer.\

- A pointer to an allocated conversion facet object, which is freed when the `wbuffer_convert` is destroyed.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

wstring_convert Class

Article • 08/03/2021

The class template `wstring_convert` performs conversions between a wide string and a byte string.

Syntax

```
C++
```

```
template <class Codecvt, class Elem = wchar_t>
class wstring_convert
```

Parameters

Codecvt
The locale facet that represents the conversion object.

Elem
The wide-character element type.

Remarks

The class template describes an object that controls conversions between wide string objects of class `std::basic_string<Elem>` and byte string objects of class `std::basic_string<char>` (also known as `std::string`). The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a `wide_string` object) and multibyte sequences (stored in a `byte_string` object) is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

An object of this class template stores:

- A byte string to display on errors
- A wide string to display on errors
- A pointer to the allocated conversion object (which is freed when the wbuffer_convert object is destroyed)

- A conversion state object of type state_type
- A conversion count

Constructors

Constructor	Description
wstring_convert	Constructs an object of type `wstring_convert`.

Typedefs

Type name	Description
byte_string	A type that represents a byte string.
wide_string	A type that represents a wide string.
state_type	A type that represents the conversion state.
int_type	A type that represents an integer.

Member functions

Member function	Description
from_bytes	Converts a byte string to a wide string.
to_bytes	Converts a wide string to a byte string.
converted	Returns the number of successful conversions.
state	Returns an object representing the state of the conversion.

Requirements

Header: <locale>

Namespace: std

wstring_convert::byte_string

A type that represents a byte string.

```
C++
```

```
typedef std::basic_string<char> byte_string;
```

Remarks

The type is a synonym for `std::basic_string<char>`.

wstring_convert::converted

Returns the number of successful conversions.

```
C++
```

```
size_t converted() const;
```

Return Value

The number of successful conversions.

Remarks

The number of successful conversions is stored in the conversion count object.

wstring_convert::from_bytes

Converts a byte string to a wide string.

```
C++
```

```
wide_string from_bytes(char Byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& Bstr);
wide_string from_bytes(const char* first, const char* last);
```

Parameters

Byte
The single-element byte sequence to be converted.

ptr
The C-style, null-terminated sequence of characters to be converted.

Bstr
The byte_string to be converted.

first
The first character in a range of characters to be converted.

last
The last character in a range of characters to be converted.

Return Value

A wide string object resulting from the conversion.

Remarks

If the conversion state object was *not* constructed with an explicit value, it is set to its default value (the initial conversion state) before the conversion begins. Otherwise it is left unchanged.

The number of input elements successfully converted is stored in the conversion count object. If no conversion error occurs, the member function returns the converted wide string. Otherwise, if the object was constructed with an initializer for the wide-string error message, the member function returns the wide-string error message object. Otherwise, the member function throws an object of class range_error.

wstring_convert::int_type

A type that represents an integer.

```
C++

typedef typename wide_string::traits_type::int_type int_type;
```

Remarks

The type is a synonym for `wide_string::traits_type::int_type`.

wstring_convert::state

Returns an object representing the state of the conversion.

```
C++
```

```
state_type state() const;
```

Return Value

The conversion state object that represents the state of the conversion.

Remarks

wstring_convert::state_type

A type that represents the conversion state.

```
C++
```

```
typedef typename Codecvt::state_type state_type;
```

Remarks

The type describes an object that can represent a conversion state. The type is a synonym for `Codecvt::state_type`.

wstring_convert::to_bytes

Converts a wide string to a byte string.

```
C++
```

```
byte_string to_bytes(Elem Char);
byte_string to_bytes(const Elem* Wptr);
byte_string to_bytes(const wide_string& Wstr);
byte_string to_bytes(const Elem* first, const Elem* last);
```

Parameters

Char
The wide character to be converted.

Wptr
The C-style, null-terminated sequence, beginning at `wptr`, to be converted.

Wstr
The wide_string to be converted.

first
The first element in a range of elements to be converted.

last
The last element in a range of elements to be converted.

Remarks

If the conversion state object was *not* constructed with an explicit value, it is set to its default value (the initial conversion state) before the conversion begins. Otherwise it is left unchanged.

The number of input elements successfully converted is stored in the conversion count object. If no conversion error occurs, the member function returns the converted byte string. Otherwise, if the object was constructed with an initializer for the byte-string error message, the member function returns the byte-string error message object. Otherwise, the member function throws an object of class range_error.

wstring_convert::wide_string

A type that represents a wide string.

```
C++
```

```
typedef std::basic_string<Elem> wide_string;
```

Remarks

The type is a synonym for `std::basic_string<Elem>`.

wstring_convert::wstring_convert

Constructs an object of type `wstring_convert`.

```
C++
```

```
wstring_convert(Codecvt *Pcvt = new Codecvt);
wstring_convert(Codecvt *Pcvt, state_type _State);
wstring_convert(const byte_string& _Berr, const wide_string& Werr =
wide_string());
```

Parameters

**Pcvt*

The object of type `Codecvt` to perform the conversion.

_State

The object of type state_type representing the conversion state.

_Berr

The byte_string to display on errors.

Werr

The wide_string to display on errors.

Remarks

The first constructor stores *Pcvt_arg* in the conversion object

<map>

Article • 12/07/2021

Defines the container class templates map and multimap and their supporting templates.

Requirements

Header: <map>

Namespace: std

> **Note**
>
> The <map> library also uses the `#include <initializer_list>` statement.

Members

Operators

Map version	Multimap version	Description
operator!= (map)	operator!= (multimap)	Tests if the map or multimap object on the left side of the operator is not equal to the map or multimap object on the right side.
operator< (map)	operator< (multimap)	Tests if the map or multimap object on the left side of the operator is less than the map or multimap object on the right side.
operator<= (map)	operator<= (multimap)	Tests if the map or multimap object on the left side of the operator is less than or equal to the map or multimap object on the right side.
operator== (map)	operator== (multimap)	Tests if the map or multimap object on the left side of the operator is equal to the map or multimap object on the right side.
operator> (map)	operator> (multimap)	Tests if the map or multimap object on the left side of the operator is greater than the map or multimap object on the right side.
operator>= (map)	operator>= (multimap)	Tests if the map or multimap object on the left side of the operator is greater than or equal to the map or multimap object on the right side.

Specialized Template Functions

Map version	Multimap version	Description
swap (map)	swap (multimap)	Exchanges the elements of two maps or multimaps.

Classes

Name	Description
value_compare Class	Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.
map Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a unique key with which the data is automatically ordered.
multimap Class	Used for the storage and retrieval of data from a collection in which the each of the elements has a key with which the data is automatically ordered and the keys do not need to have unique values.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<map>` functions

Article • 12/07/2021

swap (map)

Exchanges the elements of two maps.

```
C++
```

```
template <class key, class T, class _Pr, class _Alloc>
void swap(
    map<Key, Traits, Compare, Alloctor>& left,
    map<Key, Traits, Compare, Alloctor>& right);
```

Parameters

right
The map providing the elements to be swapped, or the map whose elements are to be exchanged with those of the map *left*.

left
The map whose elements are to be exchanged with those of the map *right*.

Remarks

The template function is an algorithm specialized on the container class map to execute the member function `left`.swap(`right`). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **`template`** < **class T**> **void swap**(**T&**, **T&**), in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function map::swap for an example that uses the template version of `swap`.

swap (multimap)

Exchanges the elements of two multimaps.

```
C++
```

```
template <class key, class T, class _Pr, class _Alloc>
void swap(
    multimap<Key, Traits, Compare, Alloctor>& left,
    multimap<Key, Traits, Compare, Alloctor>& right);
```

Parameters

right
The multimap providing the elements to be swapped, or the multimap whose elements are to be exchanged with those of the multimap *left*.

left
The multimap whose elements are to be exchanged with those of the multimap *right*.

Remarks

The template function is an algorithm specialized on the container class map to execute on the container class multimap to execute the member function `left`.swap (`right`). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **`template`** < **class T**> **void swap**(**T&**, **T&**), in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function multimap::swap for an example that uses the template version of `swap`.

`<map>` operators

Article • 12/07/2021

operator!=

Tests if the map object on the left side of the operator is not equal to the map object on the right side.

C++

```
bool operator!=(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the maps are not equal; **`false`** if maps are equal.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// map_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 != m2 )
      cout << "The maps m1 and m2 are not equal." << endl;
   else
      cout << "The maps m1 and m2 are equal." << endl;

   if ( m1 != m3 )
      cout << "The maps m1 and m3 are not equal." << endl;
   else
      cout << "The maps m1 and m3 are equal." << endl;
}
```

Output

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

`operator<`

Tests if the map object on the left side of the operator is less than the map object on the right side.

C++

```
bool operator<(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the map on the left side of the operator is strictly less than the map on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// map_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map<int, int> m1, m2, m3;
   int i;
   typedef pair<int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 < m2 )
      cout << "The map m1 is less than the map m2." << endl;
   else
      cout << "The map m1 is not less than the map m2." << endl;

   if ( m1 < m3 )
      cout << "The map m1 is less than the map m3." << endl;
   else
      cout << "The map m1 is not less than the map m3." << endl;
}
```

Output

```
The map m1 is less than the map m2.
The map m1 is not less than the map m3.
```

`operator<=`

Tests if the map object on the left side of the operator is less than or equal to the map object on the right side.

C++

```
bool operator<=(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the map on the left side of the operator is less than or equal to the map on the right side of the operator; otherwise **`false`**.

Example

C++

```
// map_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3, m4;
   int i;
```

```
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 <= m2 )
      cout << "The map m1 is less than or equal to the map m2." << endl;
   else
      cout << "The map m1 is greater than the map m2." << endl;

   if ( m1 <= m3 )
      cout << "The map m1 is less than or equal to the map m3." << endl;
   else
      cout << "The map m1 is greater than the map m3." << endl;

   if ( m1 <= m4 )
      cout << "The map m1 is less than or equal to the map m4." << endl;
   else
      cout << "The map m1 is greater than the map m4." << endl;
}
```

Output

```
The map m1 is less than or equal to the map m2.
The map m1 is greater than the map m3.
The map m1 is less than or equal to the map m4.
```

operator==

Tests if the map object on the left side of the operator is equal to the map object on the right side.

C++

```
bool operator==(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the map on the left side of the operator is equal to the map on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. Two maps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// map_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 == m2 )
      cout << "The maps m1 and m2 are equal." << endl;
   else
      cout << "The maps m1 and m2 are not equal." << endl;

   if ( m1 == m3 )
      cout << "The maps m1 and m3 are equal." << endl;
   else
      cout << "The maps m1 and m3 are not equal." << endl;
}
```

Output

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

`operator>`

Tests if the map object on the left side of the operator is greater than the map object on the right side.

C++

```
bool operator>(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the map on the left side of the operator is greater than the map on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between map objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// map_op_gt.cpp
// compile with: /EHsc
```

```
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 > m2 )
      cout << "The map m1 is greater than the map m2." << endl;
   else
      cout << "The map m1 is not greater than the map m2." << endl;

   if ( m1 > m3 )
      cout << "The map m1 is greater than the map m3." << endl;
   else
      cout << "The map m1 is not greater than the map m3." << endl;
}
/* Output:
The map m1 is not greater than the map m2.
The map m1 is greater than the map m3.
*/
```

`operator>=`

Tests if the map object on the left side of the operator is greater than or equal to the map object on the right side.

C++

```
bool operator>=(
      const map <Key, Type, Traits, Allocator>& left,
      const map <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `map`.

right
An object of type `map`.

Return Value

`true` if the map on the left side of the operator is greater than or equal to the map on the right side of the list; otherwise **`false`**.

Example

C++

```
// map_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map < int, int > m1, m2, m3, m4;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 >= m2 )
      cout << "Map m1 is greater than or equal to map m2." << endl;
   else
      cout << "The map m1 is less than the map m2." << endl;

   if ( m1 >= m3 )
      cout << "Map m1 is greater than or equal to map m3." << endl;
   else
      cout << "The map m1 is less than the map m3." << endl;

   if ( m1 >= m4 )
      cout << "Map m1 is greater than or equal to map m4." << endl;
   else
      cout << "The map m1 is less than the map m4." << endl;
}
```

Output

```
The map m1 is less than the map m2.
Map m1 is greater than or equal to map m3.
Map m1 is greater than or equal to map m4.
```

operator!= (multimap)

Tests if the multimap object on the left side of the operator is not equal to the multimap object on the right side.

C++

```
bool operator!=(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimaps are not equal; **`false`** if multimaps are equal.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// multimap_op_ne.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3;
   int i;
   typedef pair <int, int> Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 != m2 )
      cout << "The multimaps m1 and m2 are not equal." << endl;
   else
      cout << "The multimaps m1 and m2 are equal." << endl;

   if ( m1 != m3 )
      cout << "The multimaps m1 and m3 are not equal." << endl;
   else
      cout << "The multimaps m1 and m3 are equal." << endl;
}
```

Output

```
The multimaps m1 and m2 are not equal.
The multimaps m1 and m3 are equal.
```

`operator<` (multimap)

Tests if the multimap object on the left side of the operator is less than the multimap object on the right side.

C++

```
bool operator<(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimap on the left side of the operator is strictly less than the multimap on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multimap_op_lt.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 < m2 )
      cout << "The multimap m1 is less than the multimap m2." << endl;
   else
      cout << "The multimap m1 is not less than the multimap m2." << endl;

   if ( m1 < m3 )
      cout << "The multimap m1 is less than the multimap m3." << endl;
   else
      cout << "The multimap m1 is not less than the multimap m3." << endl;
}
```

Output

```
The multimap m1 is less than the multimap m2.
The multimap m1 is not less than the multimap m3.
```

`operator<=` (multimap)

Tests if the multimap object on the left side of the operator is less than or equal to the multimap object on the right side.

C++

```
bool operator<=(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimap on the left side of the operator is less than or equal to the multimap on the right side of the operator; otherwise **`false`**.

Example

C++

```
// multimap_op_le.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3, m4;
   int i;
```

```
   typedef pair <int, int> Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 <= m2 )
      cout << "m1 is less than or equal to m2" << endl;
   else
      cout << "m1 is greater than m2" << endl;

   if ( m1 <= m3 )
      cout << "m1 is less than or equal to m3" << endl;
   else
      cout << "m1 is greater than m3" << endl;

   if ( m1 <= m4 )
      cout << "m1 is less than or equal to m4" << endl;
   else
      cout << "m1 is greater than m4" << endl;
}
```

Output

```
m1 is less than or equal to m2
m1 is greater than m3
m1 is less than or equal to m4
```

operator== (multimap)

Tests if the multimap object on the left side of the operator is equal to the multimap object on the right side.

C++

```
bool operator==(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left

An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimap on the left side of the operator is equal to the multimap on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. Two multimaps are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// multimap_op_eq.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap<int, int> m1, m2, m3;
   int i;
   typedef pair<int, int> Int_Pair;

   for (i = 0; i < 3; i++)
   {
      m1.insert(Int_Pair(i, i));
      m2.insert(Int_Pair(i, i*i));
      m3.insert(Int_Pair(i, i));
   }

   if ( m1 == m2 )
      cout << "m1 and m2 are equal" << endl;
   else
      cout << "m1 and m2 are not equal" << endl;

   if ( m1 == m3 )
      cout << "m1 and m3 are equal" << endl;
   else
      cout << "m1 and m3 are not equal" << endl;
}
```

Output

```
m1 and m2 are not equal
m1 and m3 are equal
```

`operator>` (multimap)

Tests if the multimap object on the left side of the operator is greater than the multimap object on the right side.

C++

```
bool operator>(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimap on the left side of the operator is greater than the multimap on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between multimap objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multimap_op_gt.cpp
// compile with: /EHsc
```

```
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 0 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
   }

   if ( m1 > m2 )
      cout << "The multimap m1 is greater than the multimap m2." << endl;
   else
      cout << "Multimap m1 is not greater than multimap m2." << endl;

   if ( m1 > m3 )
      cout << "The multimap m1 is greater than the multimap m3." << endl;
   else
      cout << "The multimap m1 is not greater than the multimap m3." <<
endl;
}
```

Output

```
Multimap m1 is not greater than multimap m2.
The multimap m1 is greater than the multimap m3.
```

`operator>=` (multimap)

Tests if the multimap object on the left side of the operator is greater than or equal to the multimap object on the right side.

C++

```
bool operator>=(
      const multimap <Key, Type, Traits, Allocator>& left,
      const multimap <Key, Type, Traits, Allocator>& right);
```

Parameters

left
An object of type `multimap`.

right
An object of type `multimap`.

Return Value

`true` if the multimap on the left side of the operator is greater than or equal to the multimap on the right side of the list; otherwise **`false`**.

Example

C++

```
// multimap_op_ge.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap < int, int > m1, m2, m3, m4;
   int i;
   typedef pair < int, int > Int_Pair;

   for ( i = 1 ; i < 3 ; i++ )
   {
      m1.insert ( Int_Pair ( i, i ) );
      m2.insert ( Int_Pair ( i, i * i ) );
      m3.insert ( Int_Pair ( i, i - 1 ) );
      m4.insert ( Int_Pair ( i, i ) );
   }

   if ( m1 >= m2 )
      cout << "The multimap m1 is greater than or equal to the multimap m2."
<< endl;
   else
      cout << "The multimap m1 is less than the multimap m2." << endl;

   if ( m1 >= m3 )
      cout << "The multimap m1 is greater than or equal to the multimap m3."
<< endl;
   else
      cout << "The multimap m1 is less than the multimap m3." << endl;

   if ( m1 >= m4 )
      cout << "The multimap m1 is greater than or equal to the multimap m4."
<< endl;
```

```
   else
      cout << "The multimap m1 is less than the multimap m4." << endl;
}
```

Output

```
The multimap m1 is less than the multimap m2.
The multimap m1 is greater than or equal to the multimap m3.
The multimap m1 is greater than or equal to the multimap m4.
```

`map` Class

Article • 08/17/2021

Used for the storage and retrieval of data from a collection in which each element is a pair that has both a data value and a sort key. The value of the key is unique and is used to automatically sort the data.

The value of an element in a map can be changed directly. The key value is a constant and can't be changed. Instead, key values associated with old elements must be deleted, and new key values must be inserted for new elements.

Syntax

C++

```
template <class Key,
    class Type,
    class Traits = less<Key>,
    class Allocator=allocator<pair <const Key, Type>>>
class map;
```

Parameters

`Key`

The key data type to be stored in the `map`.

`Type`

The element data type to be stored in the `map`.

`Traits`

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the `map`. This argument is optional and the binary predicate `less<Key>` is the default value.

In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the

map's allocation and deallocation of memory. This argument is optional and the default value is `allocator<pair<const Key, Type> >`.

Remarks

The C++ Standard Library map class is:

- A container of variable size that efficiently retrieves element values based on associated key values.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values according to a specified comparison function.
- Unique. because each of its elements must have a unique key.
- A pair-associative container, because its element data values are distinct from its key values.
- A class template, because the functionality it provides is generic and independent of element or key type. The data types used for elements and keys are specified as parameters in the class template together with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the insert and map class member functions have versions that take as template parameters a weaker input iterator, whose functionality requirements are fewer than those guaranteed by the class of bidirectional iterators. The different iterator concepts are related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with it must be limited by those requirements. An input iterator may be dereferenced to refer to some object and may be incremented to the next iterator in the sequence.

We recommend that you base the choice of container type on the kind of searching and inserting that is required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations do them in a worst-case time that is proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that specifically pointed to the removed elements.

We recommend that you make the map the associative container of choice when conditions that associate values with keys are satisfied by the application. A model for this kind of structure is an ordered list of uniquely occurring key words that have associated string values that provide definitions. If a word has more than one correct definition, so that key isn't unique, then a multimap would be the container of choice. If just the list of words is being stored, then a set would be the appropriate container. If multiple occurrences of the words are allowed, then a multiset would be appropriate.

The map orders the elements it controls by calling a stored function object of type key_compare. This stored object is a comparison function that is accessed by calling the key_comp method. In general, any two given elements are compared to determine whether one is less than the other or whether they're equivalent. As all elements are compared, an ordered sequence of non-equivalent elements is created.

> **Note**
>
> The comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate f(x,y) is a function object that has two argument objects x and y, and a return value of `true` or `false`. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive, and if equivalence is transitive, where two objects x and y are defined to be equivalent when both f(x,y) and f(y,x) are `false`. If the stronger condition of equality between keys replaces that of equivalence, the ordering becomes total (in the sense that all the elements are ordered with regard to one other), and the keys matched will be indiscernible from one other.
>
> In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See **Heterogeneous Lookup in Associative Containers** for more information.

Members

Constructors

Name	Description
map	Constructs a list of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other map.

Typedefs

Name	Description
allocator_type	A typedef for the `allocator` class for the map object.
const_iterator	A typedef for a bidirectional iterator that can read a `const` element in the `map`.
const_pointer	A typedef for a pointer to a `const` element in a map.
const_reference	A typedef for a reference to a `const` element stored in a map for reading and doing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the `map`.
difference_type	A signed integer typedef for the number of elements of a map in a range between elements pointed to by iterators.
iterator	A typedef for a bidirectional iterator that can read or modify any element in a map.
key_compare	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the `map`.
key_type	A typedef for the sort key stored in each element of the map.
mapped_type	A typedef for the data stored in each element of a map.
pointer	A typedef for a pointer to a `const` element in a map.
reference	A typedef for a reference to an element stored in a map.
reverse_iterator	A typedef for a bidirectional iterator that can read or modify an element in a reversed map.
size_type	An unsigned integer typedef for the number of elements in a map
value_type	A typedef for the type of object stored as an element in a map.

Member functions

Member function	Description
at	Finds an element with the specified key value.
begin	Returns an iterator that points to the first element in the `map`.

Member function	Description
cbegin	Returns a const iterator that points to the first element in the `map`.
cend	Returns a const past-the-end iterator.
clear	Erases all the elements of a `map`.
contains C++20	Check if there's an element with the specified key in the `map`.
count	Returns the number of elements in a map whose key matches the key specified in a parameter.
crbegin	Returns a const iterator that points to the first element in a reversed `map`.
crend	Returns a const iterator that points to the location after the last element in a reversed `map`.
emplace	Inserts an element constructed in place into the `map`.
emplace_hint	Inserts an element constructed in place into the `map`, with a placement hint.
empty	Returns `true` if a `map` is empty.
end	Returns the past-the-end iterator.
equal_range	Returns a pair of iterators. The first iterator in the pair points to the first element in a `map` with a key that is greater than a specified key. The second iterator in the pair points to the first element in the `map` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a map from the specified positions.
find	Returns an iterator that points to the location of an element in a `map` that has a key equal to a specified key.
get_allocator	Returns a copy of the `allocator` object that is used to construct the `map`.
insert	Inserts an element or a range of elements into the `map` at a specified position.
key_comp	Returns a copy of the comparison object that used to order keys in a `map`.
lower_bound	Returns an iterator to the first element in a `map` that has a key value that is equal to or greater than that of a specified key.
max_size	Returns the maximum length of the `map`.
rbegin	Returns an iterator that points to the first element in a reversed `map`.

Member function	Description
rend	Returns an iterator that points to the location after the last element in a reversed `map`.
size	Returns the number of elements in the `map`.
swap	Exchanges the elements of two maps.
upper_bound	Returns an iterator to the first element in a `map` that has a key value that is greater than that of a specified key.
value_comp	Retrieves a copy of the comparison object that is used to order element values in a `map`.

Operators

Name	Description
operator[]	Inserts an element into a map with a specified key value.
operator=	Replaces the elements of a map with a copy of another map.

`allocator_type`

A type that represents the allocator class for the map object.

```
C++
```

```
typedef Allocator allocator_type;
```

Example

See example for get_allocator for an example that uses `allocator_type`.

`at`

Finds an element with a specified key value.

```
C++
```

```
Type& at(const Key& key);
```

```
const Type& at(const Key& key) const;
```

Parameters

key

The key value to find.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value isn't found, then the function throws an object of class out_of_range Class.

Example

C++

```
// map_at.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

typedef std::map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// find and show elements
    std::cout << "c1.at('a') == " << c1.at('a') << std::endl;
    std::cout << "c1.at('b') == " << c1.at('b') << std::endl;
    std::cout << "c1.at('c') == " << c1.at('c') << std::endl;

    return (0);
    }
```

begin

Returns an iterator addressing the first element in the `map`.

C++

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the `map` or the location succeeding an empty map.

Example

C++

```
// map_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 0, 0 ) );
   m1.insert ( Int_Pair ( 1, 1 ) );
   m1.insert ( Int_Pair ( 2, 4 ) );

   m1_cIter = m1.begin ( );
   cout << "The first element of m1 is " << m1_cIter -> first << endl;

   m1_Iter = m1.begin ( );
   m1.erase ( m1_Iter );

   // The following 2 lines would err because the iterator is const
   // m1_cIter = m1.begin ( );
   // m1.erase ( m1_cIter );

   m1_cIter = m1.begin( );
   cout << "The first element of m1 is now " << m1_cIter -> first << endl;
}
```

Output

```
The first element of m1 is 0
The first element of m1 is now 1
```

cbegin

Returns a `const` iterator that addresses the location just beyond the last element in a range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional iterator addressing the first element in the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

`clear`

Erases all the elements of a map.

```
C++
```

```
void clear();
```

Example

The following example demonstrates the use of the `map::clear` member function.

C++

```
// map_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 4));

    i = m1.size();
    cout << "The size of the map is initially "
         << i << "." << endl;

    m1.clear();
    i = m1.size();
    cout << "The size of the map after clearing is "
         << i << "." << endl;
}
```

Output

```
The size of the map is initially 2.
The size of the map after clearing is 0.
```

`const_iterator`

A type that provides a bidirectional iterator that can read a **`const`** element in the `map`.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

The `const_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<constKey, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_iterator` `cIter` pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `cIter` `->` **`first`**, which is equivalent to (* `cIter`). **`first`**.

To access the value of the mapped datum for the element, use `cIter` `->` **`second`**, which is equivalent to (* `cIter`). **`second`**.

Example

See example for begin for an example that uses `const_iterator`.

`const_pointer`

A type that provides a pointer to a **`const`** element in a map.

```
C++

typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

`const_reference`

A type that provides a reference to a **`const`** element stored in a map for reading and doing **`const`** operations.

```
C++

typedef typename allocator_type::const_reference const_reference;
```

Example

C++

```
// map_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error as the
   // non-const_reference can't be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of first element in the map is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of first element in the map is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
```

`const_reverse_iterator`

A type that provides a bidirectional iterator that can read any `const` element in the `map`.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the map in reverse.

The `const_reverse_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `const_reverse_iterator crIter` pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `crIter -> first`, which is equivalent to (* `crIter`).`first`.

To access the value of the mapped datum for the element, use `crIter -> second`, which is equivalent to (* `crIter`).`first`.

Example

See the example for rend for an example of how to declare and use `const_reverse_iterator`.

count

Returns the number of elements in a map whose key matches a parameter-specified key.

```
C++

size_type count(const Key& key) const;
```

Parameters

key
The key value of the elements to be matched from the map.

Return Value

1 if the map contains an element whose sort key matches the parameter key; 0 if the map does not contain an element with a matching key.

Remarks

The member function returns the number of elements *x* in the range

[lower_bound(*key*), upper_bound(*key*))

which is 0 or 1 in the case of map, which is a unique associative container.

Example

The following example demonstrates the use of the `map::count` member function.

C++

```
// map_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Keys must be unique in map, so duplicates are ignored
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in m1 with a sort key of 1 is: 1.
The number of elements in m1 with a sort key of 2 is: 1.
The number of elements in m1 with a sort key of 3 is: 0.
```

`contains`

Checks if there's an element with the specified key in the `map`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

`K`

The type of the key.

`key`

The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent. See Heterogeneous lookup in associative containers for more information.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <map>
#include <string>
#include <iostream>
```

```
#include <functional>

int main()
{
    std::map<int, bool> m = {{0, true},{1, false}};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << m.contains(1) << '\n';
    std::cout << m.contains(2) << '\n';

    // call template function
    std::map<std::string, int, std::less<>> m2 = {{"ten", 10}, {"twenty",
20}, {"thirty", 30}};
    std::cout << m2.contains("ten");

    return 0;
}
```

Output

```
true
false
true
```

`crbegin`

Returns a const iterator addressing the first element in a reversed map.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed `map`.

Remarks

`crbegin` is used with a reversed `map` just as begin is used with a `map`.

With the return value of `crbegin`, the `map` object can't be modified

`crbegin` can be used to iterate through a `map` backwards.

Example

C++

```
// map_crbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crbegin( );
   cout << "The first element of the reversed map m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed map m1 is 3.
```

`crend`

Returns a const iterator that addresses the location succeeding the last element in a reversed map.

C++

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed `map`).

Remarks

`crend` is used with a reversed map just as end is used with a `map`.

With the return value of `crend`, the `map` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `map`.

The value returned by `crend` should not be dereferenced.

Example

C++

```
// map_crend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crend( );
   m1_crIter--;
   cout << "The last element of the reversed map m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed map m1 is 1.
```

`difference_type`

A signed integer type that can be used to represent the number of elements of a map in a range between elements pointed to by iterators.

C++

```
typedef allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range *[first, last)* between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random access iterators provided by a random access container such as vector.

Example

C++

```
// map_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 2, 30 ) );

   map <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
   m1_bIter = m1.begin( );
   m1_eIter = m1.end( );

   // Count the number of elements in a map
   map <int, int>::difference_type  df_count = 1;
   m1_Iter = m1.begin( );
   while ( m1_Iter != m1_eIter)
   {
```

```
        df_count++;
        m1_Iter++;
    }

    cout << "The number of elements in the map m1 is: "
         << df_count << "." << endl;
}
```

Output

```
The number of elements in the map m1 is: 4.
```

`emplace`

Inserts an element constructed in place (no copy or move operations are performed) into a map.

C++

```
template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);
```

Parameters

`args`
The arguments forwarded to construct an element to insert into the map unless it already contains an element whose value is equivalently ordered.

Return Value

A pair whose **`bool`** component is `true` if an insertion was made, and `false` if the map already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **`bool`** component is true, or to the existing element if the **`bool`** component is false.

To access the iterator component of a `pair` `pr`, use `pr.first`; to dereference it, use `*pr.first`. To access the **`bool`** component, use `pr.second`. For an example, see the sample code later in this article.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state isn't modified.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

C++

```
// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    map<int, string> m1;

    auto ret = m1.emplace(10, "ten");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
        cout << "map not modified" << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;
```

```
    ret = m1.emplace(10, "one zero");

    if (!ret.second){
        auto pr = *ret.first;
        cout << "Emplace failed, element with key 10 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
    }
    else{
        cout << "map modified, now contains ";
        print(m1);
    }
    cout << endl;
}
```

`emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

`args`

The arguments forwarded to construct an element to insert into the map unless the map already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.

`where`

The place to start searching for the correct point of insertion. (If that point immediately precedes *where*, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element with its key.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state isn't modified.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

Example

C++

```
// map_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: " << endl;

    for (const auto& p : m) {
        cout << "(" << p.first <<  "," << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    map<string, string> m1;

    // Emplace some test data
    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "map starting data: ";
    print(m1);
    cout << endl;

    // Emplace with hint
    // m1.end() should be the "next" element after this emplacement
```

```
    m1.emplace_hint(m1.end(), "Doug", "Engineering");

    cout << "map modified, now contains ";
    print(m1);
    cout << endl;
}
```

`empty`

Tests if a map is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the map is empty; `false` if the map is nonempty.

Example

C++

```
// map_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2;

   typedef pair <int, int> Int_Pair;
   m1.insert ( Int_Pair ( 1, 1 ) );

   if ( m1.empty( ) )
      cout << "The map m1 is empty." << endl;
   else
      cout << "The map m1 is not empty." << endl;

   if ( m2.empty( ) )
      cout << "The map m2 is empty." << endl;
   else
      cout << "The map m2 is not empty." << endl;
}
```

Output

```
The map m1 is not empty.
The map m2 is empty.
```

end

Returns the past-the-end iterator.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the map is empty, then `map::end() == map::begin()`.

Remarks

`end` is used to test whether an iterator has passed the end of its map.

The value returned by `end` should not be dereferenced.

For a code example, see map::find.

equal_range

Returns a pair of iterators that represent the lower_bound of the key and the upper_bound of the key.

C++

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **first**, and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second**, and to dereference the upper bound iterator, use *(`pr`. **second**).

Example

C++

```
// map_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef map <int, int, less<int> > IntMap;
   IntMap m1;
   map <int, int> :: const_iterator m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMap::const_iterator, IntMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the map m1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   m1_RcIter = m1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << m1_RcIter -> second << "," << endl
        << "matching the 2nd element of the pair"
```

```
         << " returned by equal_range( 2 )." << endl;

   p2 = m1.equal_range( 4 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
      cout << "The map m1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of map m1 with a key >= 40 is: "
           << p2.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2 in the map m1 is: 20.
The upper bound of the element with a key of 2 in the map m1 is: 30.
A direct call of upper_bound( 2 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 2 ).
The map m1 doesn't have an element with a key less than 40.
```

erase

Removes an element or a range of elements in a map from specified positions or removes elements that match a specified key.

C++

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the map.

Example

C++

```
// map_erase.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions
#include <utility>  // make_pair()

using namespace std;

using mymap = map<int, string>;

void printmap(const mymap& m) {
    for (const auto& elem : m) {
        cout << " [" << elem.first << ", " << elem.second << "]";
    }
    cout << endl << "size() == " << m.size() << endl << endl;
}

int main()
{
    mymap m1;

    // Fill in some data to test with, one at a time
    m1.insert(make_pair(1, "A"));
    m1.insert(make_pair(2, "B"));
    m1.insert(make_pair(3, "C"));
    m1.insert(make_pair(4, "D"));
    m1.insert(make_pair(5, "E"));
```

```
    cout << "Starting data of map m1 is:" << endl;
    printmap(m1);
    // The 1st member function removes an element at a given position
    m1.erase(next(m1.begin()));
    cout << "After the 2nd element is deleted, the map m1 is:" << endl;
    printmap(m1);

    // Fill in some data to test with, one at a time, using an initializer
list
    mymap m2
    {
        { 10, "Bob" },
        { 11, "Rob" },
        { 12, "Robert" },
        { 13, "Bert" },
        { 14, "Bobby" }
    };

    cout << "Starting data of map m2 is:" << endl;
    printmap(m2);
    // The 2nd member function removes elements
    // in the range [First, Last)
    m2.erase(next(m2.begin()), prev(m2.end()));
    cout << "After the middle elements are deleted, the map m2 is:" << endl;
    printmap(m2);

    mymap m3;

    // Fill in some data to test with, one at a time, using emplace
    m3.emplace(1, "red");
    m3.emplace(2, "yellow");
    m3.emplace(3, "blue");
    m3.emplace(4, "green");
    m3.emplace(5, "orange");
    m3.emplace(6, "purple");
    m3.emplace(7, "pink");

    cout << "Starting data of map m3 is:" << endl;
    printmap(m3);
    // The 3rd member function removes elements with a given Key
    mymap::size_type count = m3.erase(2);
    // The 3rd member function also returns the number of elements removed
    cout << "The number of elements removed from m3 is: " << count << "." <<
endl;
    cout << "After the element with a key of 2 is deleted, the map m3 is:"
<< endl;
    printmap(m3);
}
```

find

Returns an iterator that refers to the location of an element in a map that has a key equivalent to a specified key.

```
C++
```

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key
The key value to be matched by the sort key of an element from the map being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the `map` (`map::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the `map` whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the map object can't be modified. If the return value of `find` is assigned to an `iterator`, the map object can be modified

Example

```
C++
```

```
// compile with: /EHsc /W4 /MTd
#include <map>
#include <iostream>
#include <vector>
#include <string>
#include <utility>  // make_pair()

using namespace std;
```

```
template <typename A, typename B> void print_elem(const pair<A, B>& p) {
    cout << "(" << p.first << ", " << p.second << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    map<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting map m1 is (key, value):" << endl;
    print_collection(m1);

    vector<pair<int, string>> v;
    v.push_back(make_pair(43, "Tc"));
    v.push_back(make_pair(41, "Nb"));
    v.push_back(make_pair(46, "Pd"));
    v.push_back(make_pair(42, "Mo"));
    v.push_back(make_pair(44, "Ru"));
    v.push_back(make_pair(44, "Ru")); // attempt a duplicate

    cout << "Inserting the following vector data into m1:" << endl;
    print_collection(v);

    m1.insert(v.begin(), v.end());

    cout << "The modified map m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
    findit(m1, 6);
}
```

`get_allocator`

Returns a copy of the allocator object used to construct the map.

C++

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the map.

Remarks

Allocators for the map class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// map_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int>::allocator_type m1_Alloc;
   map <int, int>::allocator_type m2_Alloc;
   map <int, double>::allocator_type m3_Alloc;
   map <int, int>::allocator_type m4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   map <int, int> m1;
   map <int, int, allocator<int> > m2;
   map <int, double, allocator<double> > m3;

   m1_Alloc = m1.get_allocator( );
   m2_Alloc = m2.get_allocator( );
   m3_Alloc = m3.get_allocator( );

   cout << "The number of integers that can be allocated\n"
        << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

   cout << "The number of doubles that can be allocated\n"
```

```
        << "before free memory is exhausted: "
        << m3.max_size( ) <<  ".\n" << endl;

   // The following line creates a map m4
   // with the allocator of map m1.
   map <int, int> m4( less<int>( ), m1_Alloc );

   m4_Alloc = m4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable." << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable." << endl;
   }
}
```

insert

Inserts an element or a range of elements into a map.

C++

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
```

```
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to insert into the map unless it already contains an element whose key is equivalently ordered.

Where

The place to start searching for the correct point of insertion. (If that point immediately precedes *Where*, insertion can occur in amortized constant time instead of logarithmic time.)

ValTy

Template parameter that specifies the argument type that the map can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First

The position of the first element to be copied.

Last

The position just beyond the last element to be copied.

InputIterator

Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

IList

The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose **bool** component is true if an insertion was made, and false if the map already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is true, or to the existing element if the **bool** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the map or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state isn't modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and for map, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into a map that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `m`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see map::emplace and map::emplace_hint.

Example

C++

```
// map_insert.cpp
// compile with: /EHsc
#include <map>
```

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <utility>  // make_pair()

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{

    // insert single values
    map<int, int> m1;
    // call insert(const value_type&) version
    m1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    m1.insert(make_pair(2, 20));

    cout << "The original key and mapped values of m1 are:" << endl;
    print(m1);

    // intentionally attempt a duplicate, single element
    auto ret = m1.insert(make_pair(1, 111));
    if (!ret.second){
        auto pr = *ret.first;
        cout << "Insert failed, element with key value 1 already exists."
            << endl << "  The existing element is (" << pr.first << ", " <<
pr.second << ")"
            << endl;
    }
    else{
        cout << "The modified key and mapped values of m1 are:" << endl;
        print(m1);
    }
    cout << endl;

    // single element, with hint
    m1.insert(m1.end(), make_pair(3, 30));
    cout << "The modified key and mapped values of m1 are:" << endl;
    print(m1);
    cout << endl;

    // The templatized version inserting a jumbled range
    map<int, int> m2;
    vector<pair<int, int>> v;
    v.push_back(make_pair(43, 294));
```

```
    v.push_back(make_pair(41, 262));
    v.push_back(make_pair(45, 330));
    v.push_back(make_pair(42, 277));
    v.push_back(make_pair(44, 311));

    cout << "Inserting the following vector data into m2:" << endl;
    print(v);

    m2.insert(v.begin(), v.end());

    cout << "The modified key and mapped values of m2 are:" << endl;
    print(m2);
    cout << endl;

    // The templatized versions move-constructing elements
    map<int, string>  m3;
    pair<int, string> ip1(475, "blue"), ip2(510, "green");

    // single element
    m3.insert(move(ip1));
    cout << "After the first move insertion, m3 contains:" << endl;
    print(m3);

    // single element with hint
    m3.insert(m3.end(), move(ip2));
    cout << "After the second move insertion, m3 contains:" << endl;
    print(m3);
    cout << endl;

    map<int, int> m4;
    // Insert the elements from an initializer_list
    m4.insert({ { 4, 44 }, { 2, 22 }, { 3, 33 }, { 1, 11 }, { 5, 55 } });
    cout << "After initializer_list insertion, m4 contains:" << endl;
    print(m4);
    cout << endl;
}
```

`iterator`

A type that provides a bidirectional iterator that can read or modify any element in a map.

C++

```
typedef implementation-defined iterator;
```

Remarks

The iterator defined by map points to elements that are objects of value_type, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference an iterator *Iter* pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).second`.

Example

See example for begin for an example of how to declare and use `iterator`.

key_comp

Retrieves a copy of the comparison object used to order keys in a map.

```
C++

key_compare key_comp() const;
```

Return Value

Returns the function object that a map uses to order its elements.

Remarks

The stored object defines the member function

`bool operator(const Key& left, const Key& right);`

which returns **`true`** if `left` precedes and isn't equal to `right` in the sort order.

Example

```
C++

// map_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of m1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of m1."
           << endl;
   }

   map <int, int, greater<int> > m2;
   map <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of m2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of m2."
           << endl;
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.
```

`key_compare`

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `map`.

C++

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter *`Traits`*.

For more information on *`Traits`*, see the map Class topic.

Example

See example for key_comp for an example of how to declare and use `key_compare`.

`key_type`

A type that describes the sort key stored in each element of the map.

```
C++

typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter *`Key`*.

For more information on *`Key`*, see the **Remarks** section of the map Class topic.

Example

See example for value_type for an example of how to declare and use `key_type`.

`lower_bound`

Returns an iterator to the first element in a map with a key value that is equal to or greater than that of a specified key.

```
C++

iterator lower_bound(const Key& key);

const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key value to be compared with the sort key of an element from the map being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a map that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the `map` if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the map object can't be modified. If the return value of `lower_bound` is assigned to an `iterator`, the map object can be modified.

Example

C++

```
// map_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.lower_bound( 2 );
   cout << "The first element of map m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for this key, end( ) is returned
   m1_RcIter = m1. lower_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key of 4." << endl;
```

```
   else
      cout << "The element of map m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
   m1_AcIter--;
   m1_RcIter = m1. lower_bound ( m1_AcIter -> first );
   cout << "The element of m1 with a key matching "
        << "that of the last element is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

map

Constructs a map that is empty or that is a copy of all or part of some other map.

C++

```
map();

explicit map(
    const Traits& Comp);

map(
    const Traits& Comp,
    const Allocator& Al);

map(
    const map& Right);

map(
    map&& Right);

map(
    initializer_list<value_type> IList);

map(
    initializer_list<value_type> IList,
    const Traits& Comp);

map(
    initializer_list<value_type> IList,
```

```
    const Traits& Comp,
    const Allocator& Allocator);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
map(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

Al

The storage allocator class to be used for this map object, which defaults to `Allocator`.

Comp

The comparison function of type `const Traits` used to order the elements in the `map`, which defaults to `hash_compare`.

Right

The map of which the constructed set is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The initializer_list from which the elements are to be copied.

Remarks

All constructors store a type of allocator object that manages memory storage for the map and that can later be returned by calling get_allocator. The allocator parameter is

often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their map.

All constructors store a function object of type Traits that is used to establish an order among the keys of the map and that can later be returned by calling key_comp.

The first three constructors specify an empty initial map, the second specifying the type of comparison function (*`Comp`*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*`Al`*) to be used. The key word **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the map *`Right`*.

The fifth constructor specifies a copy of the map by moving *`Right`*.

The 6th, 7th, and 8th constructors use an `initializer_list` from which to copy the members.

The next three constructors copy the range `[First, Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

Example

C++

```
// map_map.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    typedef pair <int, int> Int_Pair;
    map <int, int>::iterator m1_Iter, m3_Iter, m4_Iter, m5_Iter, m6_Iter,
m7_Iter;
    map <int, int, less<int> >::iterator m2_Iter;

    // Create an empty map m0 of key type integer
    map <int, int> m0;

    // Create an empty map m1 with the key comparison
    // function of less than, then insert 4 elements
    map <int, int, less<int> > m1;
    m1.insert(Int_Pair(1, 10));
    m1.insert(Int_Pair(2, 20));
```

```
    m1.insert(Int_Pair(3, 30));
    m1.insert(Int_Pair(4, 40));

    // Create an empty map m2 with the key comparison
    // function of greater than, then insert 2 elements
    map <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a map m3 with the
    // allocator of map m1
    map <int, int>::allocator_type m1_Alloc;
    m1_Alloc = m1.get_allocator();
    map <int, int> m3(less<int>(), m1_Alloc);
    m3.insert(Int_Pair(3, 30));

    // Create a copy, map m4, of map m1
    map <int, int> m4(m1);

    // Create a map m5 by copying the range m1[ first,  last)
    map <int, int>::const_iterator m1_bcIter, m1_ecIter;
    m1_bcIter = m1.begin();
    m1_ecIter = m1.begin();
    m1_ecIter++;
    m1_ecIter++;
    map <int, int> m5(m1_bcIter, m1_ecIter);

    // Create a map m6 by copying the range m4[ first,  last)
    // and with the allocator of map m2
    map <int, int>::allocator_type m2_Alloc;
    m2_Alloc = m2.get_allocator();
    map <int, int> m6(m4.begin(), ++m4.begin(), less<int>(), m2_Alloc);

    cout << "m1 =";
    for (auto i : m1)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m2 =";
    for(auto i : m2)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m3 =";
    for (auto i : m3)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m4 =";
    for (auto i : m4)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m5 =";
    for (auto i : m5)
```

```
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m6 =";
    for (auto i : m6)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m7 by moving m5
    cout << "m7 =";
    map<int, int> m7(move(m5));
    for (auto i : m7)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m8 by copying in an initializer_list
    map<int, int> m8{ { { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 } } };
    cout << "m8: = ";
    for (auto i : m8)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m9 with an initializer_list and a comparator
    map<int, int> m9({ { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 } }, less<int>
());
    cout << "m9: = ";
    for (auto i : m9)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a map m10 with an initializer_list, a comparator, and an
allocator
    map<int, int> m10({ { 9, 9 }, { 10, 10 }, { 11, 11 }, { 12, 12 } },
less<int>(), m9.get_allocator());
    cout << "m10: = ";
    for (auto i : m10)
        cout << i.first << " " << i.second << ", ";
    cout << endl;
}
```

`mapped_type`

A type that represents the data stored in a map.

```
C++
```

```
typedef Type mapped_type;
```

Remarks

The type `mapped_type` is a synonym for the class's *Type* template parameter.

For more information on *Type*, see the map Class topic.

Example

See example for value_type for an example of how to declare and use `mapped_type`.

`max_size`

Returns the maximum length of the map.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the map.

Example

C++

```
// map_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: size_type i;

   i = m1.max_size( );
   cout << "The maximum possible length "
        << "of the map is " << i << "."
        << endl << "(Magnitude is machine specific.)";
}
```

`operator[]`

Inserts an element into a map with a specified key value.

```
C++
```

```
Type& operator[](const Key& key);

Type& operator[](Key&& key);
```

Parameters

key
The key value of the element that is to insert.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value isn't found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map `m` using `m[key] = DataValue;` where `DataValue` is the value of the `mapped_type` of the element with a key value of *key*.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

Example

```
C++
```

```
// map_op_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
```

```
   map <int, int> m1;
   map <int, int> :: iterator pIter;

   // Insert a data value of 10 with a key of 1
   // into a map using the operator[] member function
   m1[ 1 ] = 10;

   // Compare other ways to insert objects into a map
   m1.insert ( map <int, int> :: value_type ( 2, 20 ) );
   m1.insert ( cInt2Int ( 3, 30 ) );

   cout  << "The keys of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

   // If the key already exists, operator[]
   // changes the value of the datum in the element
   m1[ 2 ] = 40;

   // operator[] will also insert the value of the data
   // type's default constructor if the value is unspecified
   m1[5];

   cout  << "The keys of the mapped elements are now:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> first;
   cout << "." << endl;

   cout  << "The values of the mapped elements are now:";
   for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
      cout << " " << pIter -> second;
   cout << "." << endl;

// insert by moving key
    map<string, int> c2;
    string str("abc");
    cout << "c2[move(str)] == " << c2[move(str)] << endl;
    cout << "c2["abc"] == " << c2["abc"] << endl;

    return (0);
}
```

Output

```
The keys of the mapped elements are: 1 2 3.
The values of the mapped elements are: 10 20 30.
The keys of the mapped elements are now: 1 2 3 5.
```

```
The values of the mapped elements are now: 10 40 30 0.
c2[move(str)] == 0
c2["abc"] == 1
```

`operator=`

Replaces the elements of a map with a copy of another map.

C++

```
map& operator=(const map& right);
map& operator=(map&& right);
```

Parameters

`right`

The map being copied into the `map`.

Remarks

After erasing any existing elements in a `map`, `operator=` either copies or moves the contents of *`right`* into the map.

Example

C++

```
// map_operator_as.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
   {
   using namespace std;
   map<int, int> v1, v2, v3;
   map<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
```

```
   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
   }
```

`pointer`

A type that provides a pointer to an element in a map.

C++

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a map object.

rbegin

Returns an iterator addressing the first element in a reversed map.

C++

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed map or addressing what had been the last element in the unreversed map.

Remarks

`rbegin` is used with a reversed map just as begin is used with a map.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the map object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the map object can be modified.

`rbegin` can be used to iterate through a map backwards.

Example

C++

```
// map_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rbegin( );
   cout << "The first element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;
```

```
   // A map element can be erased by dereferencing to its key
   m1_rIter = m1.rbegin( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed map is "
        << m1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed map m1 is 3.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the first element in the reversed map is 2.
```

reference

A type that provides a reference to an element stored in a map.

C++

```
typedef typename allocator_type::reference reference;
```

Example

C++

```
// map_reference.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );
```

```
    // The following line would cause an error because the
    // non-const_reference can't be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the map is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the map is "
         << Ref2 << "." << endl;

    //The non-const_reference can be used to modify the
    //data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the map is 1.
The data value of first element in the map is 10.
The modified data value of first element is 15.
```

rend

Returns an iterator that addresses the location succeeding the last element in a reversed map.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed map (the location that had preceded the first element in the unreversed map).

Remarks

`rend` is used with a reversed map just as end is used with a map.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the map object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the map object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its map.

The value returned by `rend` should not be dereferenced.

Example

C++

```
// map_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;

   map <int, int> :: iterator m1_Iter;
   map <int, int> :: reverse_iterator m1_rIter;
   map <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "The last element of the reversed map m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a map in a forward order
   cout << "The map is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a map in a reverse order
   cout << "The reversed map is: ";
```

```
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;

   // A map element can be erased by dereferencing to its key
   m1_rIter = --m1.rend( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "After the erasure, the last element "
        << "in the reversed map is "
        << m1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed map m1 is 1.
The map is: 1 2 3 .
The reversed map is: 3 2 1 .
After the erasure, the last element in the reversed map is 2.
```

`reverse_iterator`

A type that provides a bidirectional iterator that can read or modify an element in a reversed map.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` can't modify the value of an element and is used to iterate through the map in reverse.

The `reverse_iterator` defined by map points to elements that are objects of value_type, that is of type `pair<const Key, Type>`, whose first member is the key to the element and whose second member is the mapped datum held by the element.

To dereference a `reverse_iterator` *rIter* pointing to an element in a map, use the `->` operator.

To access the value of the key for the element, use `rIter -> first`, which is equivalent to (* `rIter`). **first**. To access the value of the mapped datum for the element, use `rIter -> second`, which is equivalent to (* `rIter`). **first**.

Example

See example for rbegin for an example of how to declare and use `reverse_iterator`.

`size`

Returns the number of elements in the `map`.

C++

```
size_type size() const;
```

Return Value

The current length of the map.

Example

The following example demonstrates the use of the `map::size` member function.

C++

```
// map_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    map<int, int> m1, m2;
    map<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The map length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
```

```
    cout << "The map length is now " << i << "." << endl;
}
```

Output

```
The map length is 1.
The map length is now 2.
```

`size_type`

An unsigned integer type that can represent the number of elements in a map.

C++

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for size for an example of how to declare and use `size_type`.

swap

Exchanges the elements of two maps.

C++

```
void swap(
    map<Key, Type, Traits, Allocator>& right);
```

Parameters

`right`

The argument map providing the elements to be swapped with the target map.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two maps whose elements are being exchanged.

Example

C++

```
// map_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1, m2, m3;
   map <int, int>::iterator m1_Iter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m2.insert ( Int_Pair ( 10, 100 ) );
   m2.insert ( Int_Pair ( 20, 200 ) );
   m3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   //m2 is said to be the argument map; m1 the target map
   m1.swap( m2 );

   cout << "After swapping with m2, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( m1, m3 );

   cout << "After swapping with m3, map m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original map m1 is: 10 20 30.
After swapping with m2, map m1 is: 100 200.
After swapping with m3, map m1 is: 300.
```

upper_bound

Returns an iterator to the first element in a map that with a key having a value that is greater than that of a specified key.

```
C++
```

```
iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;
```

Parameters

key
The argument key value to be compared with the sort key value of an element from the map being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a map that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the `map` if no match is found for the key.

If the return value is assigned to a `const_iterator`, the map object can't be modified. If the return value is assigned to a `iterator`, the map object can be modified.

Example

```
C++
```

```
// map_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   map <int, int> m1;
   map <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
```

```
   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of map m1 with a key "
        << "greater than 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end is returned
   m1_RcIter = m1. upper_bound ( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The map m1 doesn't have an element "
           << "with a key greater than 4." << endl;
   else
      cout << "The element of map m1 with a key > 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the map can be found
   // using a dereferenced iterator addressing the location
   m1_AcIter = m1.begin( );
   m1_RcIter = m1. upper_bound ( m1_AcIter -> first );
   cout << "The 1st element of m1 with a key greater than\n"
        << "that of the initial element of m1 is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The first element of map m1 with a key greater than 2 is: 30.
The map m1 doesn't have an element with a key greater than 4.
The 1st element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

`value_comp`

The member function returns a function object that determines the order of elements in a map by comparing their key values.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a map uses to order its elements.

Remarks

For a map *m*, if two elements *e1*(*k1*, *d1*) and *e2*(*k2*, *d2*) are objects of type `value_type`, where *k1* and *k1* are their keys of type `key_type` and *d1* and *d2* are their data of type `mapped_type`, then `m.value_comp(e1, e2)` is equivalent to `m.key_comp(k1, k2)`. A stored object defines the member function

`bool operator( value_type& left, value_type& right);`

which returns **`true`** if the key value of `left` precedes and isn't equal to the key value of `right` in the sort order.

Example

C++

```
// map_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   map <int, int, less<int> > m1;
   map <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
   pair< map<int,int>::iterator, bool > pr1, pr2;

   pr1= m1.insert ( map <int, int> :: value_type ( 1, 10 ) );
   pr2= m1.insert ( map <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *pr1.first, *pr2.first ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does not precede the element ( 2,5 )."
           << endl;
   }

   if(vc1( *pr2.first, *pr1.first ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
   {
      cout << "The element ( 2,5 ) does not precede the element ( 1,10 )."
           << endl;
```

```
   }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

value_type

The type of object stored as an element in a map.

C++

```
typedef pair<const Key, Type> value_type;
```

Example

C++

```
// map_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   map <int, int> m1;
   map <int, int> :: key_type key1;
   map <int, int> :: mapped_type mapped1;
   map <int, int> :: value_type value1;
   map <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitly to avoid implicit type conversion
   m1.insert ( map <int, int> :: value_type ( 1, 10 ) );

   // Compare other ways to insert objects into a map
   m1.insert ( cInt2Int ( 2, 20 ) );
   m1[ 3 ] = 30;

   // Initializing key1 and mapped1
   key1 = ( m1.begin( ) -> first );
   mapped1 = ( m1.begin( ) -> second );
```

```
    cout << "The key of first element in the map is "
         << key1 << "." << endl;

    cout << "The data value of first element in the map is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type isn't assignable
    // value1 = cInt2Int ( 4, 40 );

    cout  << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
}
```

See also

Containers

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`multimap` Class

Article • 02/11/2022

The C++ Standard Library multimap class is used for the storage and retrieval of data from a collection in which each element is a pair that has both a data value and a sort key. The value of the key doesn't need to be unique and is used to order the data automatically. The value of an element in a multimap, but not its associated key value, may be changed directly. Instead, key values associated with old elements must be deleted and new key values associated with new elements inserted.

Syntax

C++

```
template <class Key,
    class Type,
    class Traits=less <Key>,
    class Allocator=allocator <pair  <const Key, Type>>>
class multimap;
```

Parameters

`Key`

The key data type to be stored in the multimap.

`Type`

The element data type to be stored in the multimap.

`Traits`

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the multimap. The binary predicate `less<Key>` is the default value.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see Heterogeneous Lookup in Associative Containers

`Allocator`

The type that represents the stored allocator object that encapsulates details about the map's allocation and deallocation of memory. This argument is optional and the default value is `allocator<pair <const Key, Type> >`.

Remarks

The C++ Standard Library multimap class is

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple, because its elements don't need to have a unique key, so that one key value may have many element data values associated with it.
- A pair associative container, because its element data values are distinct from its key values.
- A class template, because the functionality it provides is generic and so independent of the specific type of data contained as elements or keys. The data types to be used for elements and keys are, instead, specified as parameters in the class template along with the comparison function and allocator.

The iterator provided by the map class is a bidirectional iterator, but the class member functions insert and multimap have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators `[First, Last)` in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had pointed at the removed elements.

The `multimap` should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. A model for this type of structure is an ordered list of key words with associated string values providing, say, definitions, where the words weren't always uniquely defined. If, instead, the key words were uniquely defined so that keys were unique, then a map would be the container of choice. If, on the other hand, just the list of words were being stored, then a set would be the correct container. If multiple occurrences of the words were allowed, then a `multiset` would be the appropriate container structure.

The `multimap` orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate `f(x,y)` is a function object that has two argument objects `x` and `y` and a return value of `true` or `false`. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects `x` and `y` are defined to be equivalent when both `f(x,y)` and `f(y,x)` are `false`. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

Members

Constructors

Constructor	Description
multimap	Constructs a `multimap` that is empty or that is a copy of all or part of some other `multimap`.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for the `multimap` object.
const_iterator	A type that provides a bidirectional iterator that can read a `const` element in the `multimap`.
const_pointer	A type that provides a pointer to a `const` element in a `multimap`.
const_reference	A type that provides a reference to a `const` element stored in a `multimap` for reading and performing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the `multimap`.
difference_type	A signed integer type that can be used to represent the number of elements of a `multimap` in a range between elements pointed to by iterators.
iterator	A type that provides the difference between two iterators that refer to elements within the same `multimap`.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `multimap`.
key_type	A type that describes the sort key object that makes up each element of the `multimap`.
mapped_type	A type that represents the data type stored in a `multimap`.
pointer	A type that provides a pointer to a `const` element in a `multimap`.
reference	A type that provides a reference to an element stored in a `multimap`.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed `multimap`.
size_type	An unsigned integer type that provides a pointer to a `const` element in a `multimap`.
value_type	A type that provides a function object that can compare two elements as sort keys to determine their relative order in the `multimap`.

Member functions

Member function	Description

Member function	Description
begin	Returns an iterator addressing the first element in the `multimap`.
cbegin	Returns a `const` iterator addressing the first element in the `multimap`.
cend	Returns a `const` iterator that addresses the location succeeding the last element in a `multimap`.
clear	Erases all the elements of a `multimap`.
contains C++20	Checks if there is an element with the specified key in the `multimap`.
count	Returns the number of elements in a `multimap` whose key matches a parameter-specified key.
crbegin	Returns a `const` iterator addressing the first element in a reversed `multimap`.
crend	Returns a `const` iterator that addresses the location succeeding the last element in a reversed `multimap`.
emplace	Inserts an element constructed in place into a `multimap`.
emplace_hint	Inserts an element constructed in place into a `multimap`, with a placement hint
empty	Tests if a `multimap` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `multimap`.
equal_range	Finds the range of elements where the key of the element matches a specified value.
erase	Removes an element or a range of elements in a `multimap` from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the first location of an element in a `multimap` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `multimap`.
insert	Inserts an element or a range of elements into a `multimap`.
key_comp	Retrieves a copy of the comparison object used to order keys in a `multimap`.
lower_bound	Returns an iterator to the first element in a `multimap` that with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `multimap`.
rbegin	Returns an iterator addressing the first element in a reversed `multimap`.

Member function	Description
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `multimap`.
size	Returns the number of elements in the `multimap`.
swap	Exchanges the elements of two `multimap`s.
upper_bound	Returns an iterator to the first element in a `multimap` that with a key that is greater than a specified key.
value_comp	The member function returns a function object that determines the order of elements in a `multimap` by comparing their key values.

Operator	Description
operator=	Replaces the elements of a `multimap` with a copy of another `multimap`.

Requirements

Header: `<map>`

Namespace: `std`

The (**key**, **value**) pairs are stored in a `multimap` as objects of type `pair`. The pair class requires the header `<utility>`, which is automatically included by `<map>`.

`multimap::allocator_type`

A type that represents the allocator class for the `multimap` object.

```
C++

typedef Allocator allocator_type;
```

Example

See the example for get_allocator for an example using `allocator_type`.

`multimap::begin`

Returns an iterator addressing the first element in the `multimap`.

```
C++
```

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the `multimap` or the location succeeding an empty `multimap`.

Example

```
C++
```

```
// multimap_begin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: const_iterator m1_cIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 0, 0 ) );
   m1.insert ( Int_Pair ( 1, 1 ) );
   m1.insert ( Int_Pair ( 2, 4 ) );

   m1_cIter = m1.begin ( );
   cout << "The first element of m1 is " << m1_cIter -> first << endl;

   m1_Iter = m1.begin ( );
   m1.erase ( m1_Iter );

   // The following 2 lines would err as the iterator is const
   // m1_cIter = m1.begin ( );
   // m1.erase ( m1_cIter );

   m1_cIter = m1.begin( );
   cout << "First element of m1 is now " << m1_cIter -> first << endl;
}
```

Output

```
The first element of m1 is 0
First element of m1 is now 1
```

`multimap::cbegin`

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`multimap::cend`

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`multimap::clear`

Erases all the elements of a `multimap`.

```
C++
```

```
void clear();
```

Example

The following example demonstrates the use of the `multimap::clear` member function.

C++

```
// multimap_clear.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap<int, int> m1;
   multimap<int, int>::size_type i;
   typedef pair<int, int> Int_Pair;

   m1.insert(Int_Pair(1, 1));
   m1.insert(Int_Pair(2, 4));

   i = m1.size();
   cout << "The size of the multimap is initially "
        << i << "." << endl;

   m1.clear();
   i = m1.size();
   cout << "The size of the multimap after clearing is "
        << i << "." << endl;
}
```

Output

```
The size of the multimap is initially 2.
The size of the multimap after clearing is 0.
```

`multimap::const_iterator`

A type that provides a bidirectional iterator that can read a **`const`** element in the `multimap`.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

The `const_iterator` defined by `multimap` points to objects of value_type, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_iterator` *`cIter`* pointing to an element in a `multimap`, use the `->` operator.

To access the value of the key for the element, use `cIter->first`, which is equivalent to `(*cIter).first`. To access the value of the mapped datum for the element, use `cIter->second`, which is equivalent to `(*cIter).second`.

Example

See the example for begin for an example using `const_iterator`.

`multimap::const_pointer`

A type that provides a pointer to a **`const`** element in a `multimap`.

```
C++

typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `multimap` object.

`multimap::const_reference`

A type that provides a reference to a **`const`** element stored in a `multimap` for reading and performing **`const`** operations.

```
C++

typedef typename allocator_type::const_reference const_reference;
```

Example

C++

```
// multimap_const_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );

   // The following line would cause an error because the
   // non-const_reference can't be used to access the key
   // int &Ref1 = ( m1.begin( ) -> first );

   cout << "The key of the first element in the multimap is "
        << Ref1 << "." << endl;

   // Declare and initialize a reference &Ref2
   // to the data value of the first element
   int &Ref2 = ( m1.begin( ) -> second );

   cout << "The data value of the first element in the multimap is "
        << Ref2 << "." << endl;
}
```

Output

```
The key of the first element in the multimap is 1.
The data value of the first element in the multimap is 10.
```

`multimap::const_reverse_iterator`

A type that provides a bidirectional iterator that can read any `const` element in the `multimap`.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the `multimap` in reverse.

The `const_reverse_iterator` defined by `multimap` points to objects of value_type, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `const_reverse_iterator` *`crIter`* pointing to an element in a multimap, use the `->` operator.

To access the value of the key for the element, use `crIter->first`, which is equivalent to `(*crIter).first`. To access the value of the mapped datum for the element, use `crIter->second`, which is equivalent to `(*crIter).first`.

Example

See the example for rend for an example of how to declare and use `const_reverse_iterator`.

`multimap::contains`

Check if there is an element with the specified key in the `multimap`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

`K`

The type of the key.

`key`

The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent. See Heterogeneous lookup in associative containers for more information.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <map>
#include <string>
#include <iostream>
#include <functional>

int main()
{
    std::multimap<int, bool> m = {{0, false}, {1, true}};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << m.contains(1) << '\n';
    std::cout << m.contains(2) << '\n';

    // call template function
    std::multimap<std::string, int, std::less<>> m2 = {{"ten", 10},
{"twenty", 20}, {"thirty", 30}};
    std::cout << m2.contains("ten");

    return 0;
}
```

Output

```
true
false
true
```

`multimap::count`

Returns the number of elements in a `multimap` whose keys match a parameter-specified key.

```
C++
```

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the multimap.

Return Value

The number of elements whose sort keys match the parameter key; 0 if the multimap doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the range

lower_bound(key), upper_bound(key)

that have a key value *key*.

Example

The following example demonstrates the use of the multimap::count member function.

```
C++
```

```
// multimap_count.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
    using namespace std;
    multimap<int, int> m1;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    m1.insert(Int_Pair(2, 1));
```

```
    m1.insert(Int_Pair(1, 4));
    m1.insert(Int_Pair(2, 1));

    // Elements don't need to have unique keys in multimap,
    // so duplicates are allowed and counted
    i = m1.count(1);
    cout << "The number of elements in m1 with a sort key of 1 is: "
         << i << "." << endl;

    i = m1.count(2);
    cout << "The number of elements in m1 with a sort key of 2 is: "
         << i << "." << endl;

    i = m1.count(3);
    cout << "The number of elements in m1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in m1 with a sort key of 1 is: 2.
The number of elements in m1 with a sort key of 2 is: 2.
The number of elements in m1 with a sort key of 3 is: 0.
```

`multimap::crbegin`

Returns a `const` iterator addressing the first element in a reversed `multimap`.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A `const` reverse bidirectional iterator addressing the first element in a reversed multimap or addressing what had been the last element in the unreversed `multimap`.

Remarks

`crbegin` is used with a reversed `multimap` just as begin is used with a `multimap`.

With the return value of `crbegin`, the `multimap` object can't be modified.

`crbegin` can be used to iterate through a `multimap` backwards.

Example

C++

```
// multimap_crbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crbegin( );
   cout << "The first element of the reversed multimap m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed multimap m1 is 3.
```

`multimap::crend`

Returns a `const` iterator that addresses the location succeeding the last element in a reversed `multimap`.

C++

```
const_reverse_iterator crend() const;
```

Return Value

A `const` reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multimap (the location that had preceded the first element in the unreversed `multimap`).

Remarks

`crend` is used with a reversed `multimap` just as multimap::end is used with a `multimap`.

With the return value of `crend`, the `multimap` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `multimap`.

The value returned by `crend` shouldn't be dereferenced.

Example

C++

```
// multimap_crend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_crIter = m1.crend( );
   m1_crIter--;
   cout << "The last element of the reversed multimap m1 is "
        << m1_crIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed multimap m1 is 1.
```

`multimap::difference_type`

A signed integer type that can be used to represent the number of elements of a `multimap` in a range between elements pointed to by iterators.

C++

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [*first*, *last*) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

C++

```
// multimap_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <map>
#include <algorithm>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );

   // The following will insert as multimap keys are not unique
   m1.insert ( Int_Pair ( 2, 30 ) );

   multimap <int, int>::iterator m1_Iter, m1_bIter, m1_eIter;
   m1_bIter = m1.begin( );
   m1_eIter = m1.end( );

   // Count the number of elements in a multimap
   multimap <int, int>::difference_type  df_count = 0;
   m1_Iter = m1.begin( );
```

```
   while ( m1_Iter != m1_eIter )
   {
      df_count++;
      m1_Iter++;
   }

   cout << "The number of elements in the multimap m1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number of elements in the multimap m1 is: 4.
```

`multimap::emplace`

Inserts an element constructed in place (no copy or move operations are performed).

C++

```
template <class... Args>
iterator emplace(Args&&... args);
```

Parameters

`args`

The arguments forwarded to construct an element to be inserted into the `multimap`.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

If an exception is thrown during the insertion, the container is left unaltered and the exception is rethrown.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component

equal to the data value of the element.

Example

C++

```
// multimap_emplace.cpp
// compile with: /EHsc
#include <map>
#include <string>
#include <iostream>

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: " << endl;

    for (const auto& p : m) {
        cout << "(" << p.first <<  "," << p.second << ") ";
    }

    cout << endl;
}

int main()
{
    multimap<string, string> m1;

    m1.emplace("Anna", "Accounting");
    m1.emplace("Bob", "Accounting");
    m1.emplace("Carmine", "Engineering");

    cout << "multimap modified, now contains ";
    print(m1);
    cout << endl;

    m1.emplace("Bob", "Engineering");

    cout << "multimap modified, now contains ";
    print(m1);
    cout << endl;
}
```

`multimap::emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

args

The arguments forwarded to construct an element to be inserted into the `multimap`.

where

The place to start searching for the correct point of insertion. (If that point immediately precedes *where*, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state is not modified.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see map::emplace_hint.

`multimap::empty`

Tests if a `multimap` is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the `multimap` is empty; `false` if the `multimap` is nonempty.

Example

C++

```
// multimap_empty.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2;

   typedef pair <int, int> Int_Pair;
   m1.insert ( Int_Pair ( 1, 1 ) );

   if ( m1.empty( ) )
      cout << "The multimap m1 is empty." << endl;
   else
      cout << "The multimap m1 is not empty." << endl;

   if ( m2.empty( ) )
      cout << "The multimap m2 is empty." << endl;
   else
      cout << "The multimap m2 is not empty." << endl;
}
```

Output

```
The multimap m1 is not empty.
The multimap m2 is empty.
```

`multimap::end`

Returns the past-the-end iterator.

C++

```
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the `multimap` is empty, then `multimap::end() == multimap::begin()`.

Remarks

`end` is used to test whether an iterator has passed the end of its `multimap`.

The value returned by **`end`** shouldn't be dereferenced.

For a code example, see multimap::find.

`multimap::equal_range`

Finds the range of elements where the key of the element matches a specified value.

```
C++

pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the `multimap` being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr.first` and to dereference the lower bound iterator, use `*(pr.first)`. To access the second iterator of a pair `pr` returned by the member function, use `pr.second` and to dereference the upper bound iterator, use `*(pr.second)`.

Example

```
C++
```

```cpp
// multimap_equal_range.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef multimap <int, int, less<int> > IntMMap;
   IntMMap m1;
   multimap <int, int> :: const_iterator m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   pair <IntMMap::const_iterator, IntMMap::const_iterator> p1, p2;
   p1 = m1.equal_range( 2 );

   cout << "The lower bound of the element with "
        << "a key of 2 in the multimap m1 is: "
        << p1.first -> second << "." << endl;

   cout << "The upper bound of the element with "
        << "a key of 2 in the multimap m1 is: "
        << p1.second -> second << "." << endl;

   // Compare the upper_bound called directly
   m1_RcIter = m1.upper_bound( 2 );

   cout << "A direct call of upper_bound( 2 ) gives "
        << m1_RcIter -> second << "," << endl
        << "matching the 2nd element of the pair "
        << "returned by equal_range( 2 )." << endl;

   p2 = m1.equal_range( 4 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == m1.end( ) ) && ( p2.second == m1.end( ) ) )
      cout << "The multimap m1 doesn't have an element "
           << "with a key less than 4." << endl;
   else
      cout << "The element of multimap m1 with a key >= 40 is: "
           << p1.first -> first << "." << endl;
}
```

Output

```
The lower bound of the element with a key of 2 in the multimap m1 is: 20.
The upper bound of the element with a key of 2 in the multimap m1 is: 30.
```

```
A direct call of upper_bound( 2 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 2 ).
The multimap m1 doesn't have an element with a key less than 4.
```

`multimap::erase`

Removes an element or a range of elements in a `multimap` from specified positions or removes elements that match a specified key.

C++

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

`Where`
Position of the element to be removed.

`First`
Position of the first element to be removed.

`Last`
Position just beyond the last element to be removed.

`Key`
The key of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the `multimap`.

Remarks

For a code example, see map::erase.

`multimap::find`

Returns an iterator that refers to the first location of an element in a `multimap` that has a key equivalent to a specified key.

```
C++
```

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

`key`
The key value to be matched by the sort key of an element from the `multimap` being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the `multimap` (`multimap::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the `multimap` whose sort key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the `multimap` object can't be modified. If the return value of `find` is assigned to an `iterator`, the `multimap` object can be modified.

Example

```
C++
```

```cpp
// compile with: /EHsc /W4 /MTd
#include <map>
#include <iostream>
#include <vector>
#include <string>
#include <utility>  // make_pair()

using namespace std;

template <typename A, typename B> void print_elem(const pair<A, B>& p) {
    cout << "(" << p.first << ", " << p.second << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    multimap<int, string> m1({ { 40, "Zr" }, { 45, "Rh" } });
    cout << "The starting multimap m1 is (key, value):" << endl;
    print_collection(m1);

    vector<pair<int, string>> v;
    v.push_back(make_pair(43, "Tc"));
    v.push_back(make_pair(41, "Nb"));
    v.push_back(make_pair(46, "Pd"));
    v.push_back(make_pair(42, "Mo"));
    v.push_back(make_pair(44, "Ru"));
    v.push_back(make_pair(44, "Ru")); // attempt a duplicate

    cout << "Inserting the following vector data into m1:" << endl;
    print_collection(v);

    m1.insert(v.begin(), v.end());

    cout << "The modified multimap m1 is (key, value):" << endl;
    print_collection(m1);
    cout << endl;
    findit(m1, 45);
```

```
    findit(m1, 6);
}
```

`multimap::get_allocator`

Returns a copy of the allocator object used to construct the `multimap`.

C++

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the `multimap`.

Remarks

Allocators for the `multimap` class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// multimap_get_allocator.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int>::allocator_type m1_Alloc;
   multimap <int, int>::allocator_type m2_Alloc;
   multimap <int, double>::allocator_type m3_Alloc;
   multimap <int, int>::allocator_type m4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   multimap <int, int> m1;
   multimap <int, int, allocator<int> > m2;
   multimap <int, double, allocator<double> > m3;
```

```
   m1_Alloc = m1.get_allocator( );
   m2_Alloc = m2.get_allocator( );
   m3_Alloc = m3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << m2.max_size( ) << ".\n" << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << m3.max_size( ) <<  ".\n" << endl;

   // The following line creates a multimap m4
   // with the allocator of multimap m1.
   map <int, int> m4( less<int>( ), m1_Alloc );

   m4_Alloc = m4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated via the other
   if( m1_Alloc == m4_Alloc )
   {
      cout << "The allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "The allocators are not interchangeable."
           << endl;
   }
}
```

`multimap::insert`

Inserts an element or a range of elements into a `multimap`.

C++

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
```

```
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to be inserted into the `multimap`.

Where

The place to start searching for the correct point of insertion. (If that point immediately precedes *Where*, insertion can occur in amortized constant time instead of logarithmic time.)

ValTy

Template parameter that specifies the argument type that the map can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First

The position of the first element to be copied.

Last

The position just beyond the last element to be copied.

InputIterator

Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

IList

The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the `multimap`.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `multimap`.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The value_type of a container is a typedef that belongs to the container, and for map, `multimap<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into a `multimap` that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *`Last`* doesn't get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an initializer_list to copy elements into the map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see multimap::emplace and multimap::emplace_hint.

Example

```
C++
```

```cpp
// multimap_insert.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
#include <string>
#include <vector>
#include <utility>  // make_pair()

using namespace std;

template <typename M> void print(const M& m) {
    cout << m.size() << " elements: ";

    for (const auto& p : m) {
        cout << "(" << p.first << ", " << p.second << ") ";
    }

    cout << endl;
}

int main()
{

    // insert single values
    multimap<int, int> m1;
    // call insert(const value_type&) version
    m1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    m1.insert(make_pair(2, 20));

    cout << "The original key and mapped values of m1 are:" << endl;
    print(m1);

    // intentionally attempt a duplicate, single element
    m1.insert(make_pair(1, 111));

    cout << "The modified key and mapped values of m1 are:" << endl;
    print(m1);

    // single element, with hint
    m1.insert(m1.end(), make_pair(3, 30));
    cout << "The modified key and mapped values of m1 are:" << endl;
    print(m1);
    cout << endl;

    // The templatized version inserting a jumbled range
    multimap<int, int> m2;
    vector<pair<int, int>> v;
    v.push_back(make_pair(43, 294));
    v.push_back(make_pair(41, 262));
    v.push_back(make_pair(45, 330));
    v.push_back(make_pair(42, 277));
    v.push_back(make_pair(44, 311));
```

```
    cout << "Inserting the following vector data into m2:" << endl;
    print(v);

    m2.insert(v.begin(), v.end());

    cout << "The modified key and mapped values of m2 are:" << endl;
    print(m2);
    cout << endl;

    // The templatized versions move-constructing elements
    multimap<int, string>  m3;
    pair<int, string> ip1(475, "blue"), ip2(510, "green");

    // single element
    m3.insert(move(ip1));
    cout << "After the first move insertion, m3 contains:" << endl;
    print(m3);

    // single element with hint
    m3.insert(m3.end(), move(ip2));
    cout << "After the second move insertion, m3 contains:" << endl;
    print(m3);
    cout << endl;

    multimap<int, int> m4;
    // Insert the elements from an initializer_list
    m4.insert({ { 4, 44 }, { 2, 22 }, { 3, 33 }, { 1, 11 }, { 5, 55 } });
    cout << "After initializer_list insertion, m4 contains:" << endl;
    print(m4);
    cout << endl;
}
```

`multimap::iterator`

A type that provides a bidirectional iterator that can read or modify any element in a `multimap`.

C++

```
typedef implementation-defined iterator;
```

Remarks

The `iterator` defined by `multimap` points to objects of value_type, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference an `iterator` *`Iter`* pointing to an element in a `multimap`, use the `->` operator.

To access the value of the key for the element, use `Iter->first`, which is equivalent to `(*Iter).first`. To access the value of the mapped datum for the element, use `Iter->second`, which is equivalent to `(*Iter).second`.

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin for an example of how to declare and use `iterator`.

`multimap::key_comp`

Retrieves a copy of the comparison object used to order keys in a `multimap`.

C++

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a `multimap` uses to order its elements.

Remarks

The stored object defines the member function

`bool operator( const Key& x, const Key& y);`

which returns true if *`x`* strictly precedes *`y`* in the sort order.

Example

C++

```
// multimap_key_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   multimap <int, int, less<int> > m1;
   multimap <int, int, less<int> >::key_compare kc1 = m1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of m1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of m1."
           << endl;
   }

   multimap <int, int, greater<int> > m2;
   multimap <int, int, greater<int> >::key_compare kc2 = m2.key_comp( );
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of m2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of m2."
           << endl;
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of m1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of m2.
```

`multimap::key_compare`

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `multimap`.

C++

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits`, see the multimap Class topic.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

`multimap::key_type`

A type that describes the sort key object that constitutes each element of the `multimap`.

```
C++
```

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the multimap Class topic.

Example

See the example for value_type for an example of how to declare and use `key_type`.

`multimap::lower_bound`

Returns an iterator to the first element in a `multimap` that with a key that is equal to or greater than a specified key.

```
C++
```

```
iterator lower_bound(const Key& key);

const_iterator lower_bound(const Key& key) const;
```

Parameters

key

The argument key to be compared with the sort key of an element from the `multimap` being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a `multimap` that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the `multimap` if no match is found for the key.

If the return value of `lower_bound` is assigned to a `const_iterator`, the `multimap` object can't be modified. If the return value of `lower_bound` is assigned to an **iterator**, the `multimap` object can be modified.

Example

C++

```
// multimap_lower_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_RcIter = m1.lower_bound( 2 );
   cout << "The element of multimap m1 with a key of 2 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.lower_bound( 3 );
   cout << "The first element of multimap m1 with a key of 3 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.lower_bound( 4 );
```

```
   if ( m1_RcIter == m1.end( ) )
      cout << "The multimap m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of multimap m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the multimap can be
   // found using a dereferenced iterator addressing the location
   m1_AcIter = m1.end( );
   m1_AcIter--;
   m1_RcIter = m1.lower_bound( m1_AcIter -> first );
   cout << "The first element of m1 with a key matching\n"
        << "that of the last element is: "
        << m1_RcIter -> second << "." << endl;

   // Note that the first element with a key equal to
   // the key of the last element is not the last element
   if ( m1_RcIter == --m1.end( ) )
      cout << "This is the last element of multimap m1."
           << endl;
   else
      cout << "This is not the last element of multimap m1."
           << endl;
}
```

Output

```
The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key matching
that of the last element is: 20.
This is not the last element of multimap m1.
```

`multimap::mapped_type`

A type that represents the data type stored in a `multimap`.

C++

```
typedef Type mapped_type;
```

Remarks

`mapped_type` is a synonym for the template parameter `Type`.

For more information on `Type`, see the multimap Class topic.

Example

See the example for value_type for an example of how to declare and use `key_type`.

`multimap::max_size`

Returns the maximum length of the `multimap`.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the `multimap`.

Example

C++

```
// multimap_max_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: size_type i;

   i = m1.max_size( );
   cout << "The maximum possible length "
        << "of the multimap is " << i << "." << endl;
}
```

`multimap::multimap`

Constructs a `multimap` that is empty or that is a copy of all or part of some other `multimap`.

C++

```
multimap();

explicit multimap(
    const Traits& Comp);

multimap(
    const Traits& Comp,
    const Allocator& Al);

map(
    const multimap& Right);

multimap(
    multimap&& Right);

multimap(
    initializer_list<value_type> IList);

multimap(
    initializer_list<value_type> IList,
    const Compare& Comp);

multimap(
    initializer_list<value_type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
multimap(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

Al

The storage allocator class to be used for this multimap object, which defaults to

Allocator.

Comp

The comparison function of type `constTraits` used to order the elements in the map, which defaults to `Traits`.

Right

The map of which the constructed set is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The initializer_list from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the `multimap` and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their `multimap`.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the `multimap` and that can later be returned by calling key_comp.

The first three constructors specify an empty initial `multimap`, the second specifying the type of comparison function (*Comp*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*Al*) to be used. The key word **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the `multimap` *Right*.

The fifth constructor specifies a copy of the `multimap` by moving *Right*.

The 6th, 7th, and 8th constructors copy the members of an `initializer_list`.

The next three constructors copy the range `[First, Last)` of a map with increasing explicitness in specifying the type of comparison function of class `Traits` and allocator.

Example

C++

```
// multimap_ctor.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    typedef pair <int, int> Int_Pair;

    // Create an empty multimap m0 of key type integer
    multimap <int, int> m0;

    // Create an empty multimap m1 with the key comparison
    // function of less than, then insert 4 elements
    multimap <int, int, less<int> > m1;
    m1.insert(Int_Pair(1, 10));
    m1.insert(Int_Pair(2, 20));
    m1.insert(Int_Pair(3, 30));
    m1.insert(Int_Pair(4, 40));

    // Create an empty multimap m2 with the key comparison
    // function of greater than, then insert 2 elements
    multimap <int, int, less<int> > m2;
    m2.insert(Int_Pair(1, 10));
    m2.insert(Int_Pair(2, 20));

    // Create a multimap m3 with the
    // allocator of multimap m1
    multimap <int, int>::allocator_type m1_Alloc;
    m1_Alloc = m1.get_allocator();
    multimap <int, int> m3(less<int>(), m1_Alloc);
    m3.insert(Int_Pair(3, 30));

    // Create a copy, multimap m4, of multimap m1
    multimap <int, int> m4(m1);

    // Create a multimap m5 by copying the range m1[ first,  last)
    multimap <int, int>::const_iterator m1_bcIter, m1_ecIter;
    m1_bcIter = m1.begin();
    m1_ecIter = m1.begin();
    m1_ecIter++;
    m1_ecIter++;
    multimap <int, int> m5(m1_bcIter, m1_ecIter);

    // Create a multimap m6 by copying the range m4[ first,  last)
    // and with the allocator of multimap m2
    multimap <int, int>::allocator_type m2_Alloc;
    m2_Alloc = m2.get_allocator();
    multimap <int, int> m6(m4.begin(), ++m4.begin(), less<int>(), m2_Alloc);
```

```
    cout << "m1 =";
    for (auto i : m1)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m2 =";
    for (auto i : m2)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m3 =";
    for (auto i : m3)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m4 =";
    for (auto i : m4)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m5 =";
    for (auto i : m5)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    cout << "m6 =";
    for (auto i : m6)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a multimap m8 by copying in an initializer_list
    multimap<int, int> m8{ { { 1, 1 }, { 2, 2 }, { 3, 3 }, { 4, 4 } } };
    cout << "m8: = ";
    for (auto i : m8)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a multimap m9 with an initializer_list and a comparator
    multimap<int, int> m9({ { 5, 5 }, { 6, 6 }, { 7, 7 }, { 8, 8 } },
less<int>());
    cout << "m9: = ";
    for (auto i : m9)
        cout << i.first << " " << i.second << ", ";
    cout << endl;

    // Create a multimap m10 with an initializer_list, a comparator, and an
allocator
    multimap<int, int> m10({ { 9, 9 }, { 10, 10 }, { 11, 11 }, { 12, 12 } },
less<int>(), m9.get_allocator());
    cout << "m10: = ";
    for (auto i : m10)
        cout << i.first << " " << i.second << ", ";
    cout << endl;
```

```
}
```

`multimap::operator=`

Replaces the elements of a `multimap` with a copy of another `multimap`.

C++

```
multimap& operator=(const multimap& right);

multimap& operator=(multimap&& right);
```

Parameters

`right`

The multimap being copied into the `multimap`.

Remarks

After erasing any existing elements in a `multimap`, `operator=` either copies or moves the contents of *`right`* into the `multimap`.

Example

C++

```
// multimap_operator_as.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
   {
   using namespace std;
   multimap<int, int> v1, v2, v3;
   multimap<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
```

```
   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
   }
```

`multimap::pointer`

A type that provides a pointer to an element in a `multimap`.

C++

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `multimap` object.

`multimap::rbegin`

Returns an iterator addressing the first element in a reversed `multimap`.

C++

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed `multimap` or addressing what had been the last element in the unreversed `multimap`.

Remarks

`rbegin` is used with a reversed `multimap` just as begin is used with a `multimap`.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the `multimap` object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the `multimap` object can be modified.

`rbegin` can be used to iterate through a `multimap` backwards.

Example

C++

```
// multimap_rbegin.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rbegin( );
   cout << "The first element of the reversed multimap m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a multimap in a forward order
   cout << "The multimap is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a multimap in a reverse order
   cout << "The reversed multimap is: ";
   for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
      cout << m1_rIter -> first << " ";
      cout << "." << endl;
```

```
   // A multimap element can be erased by dereferencing its key
   m1_rIter = m1.rbegin( );
   m1.erase ( m1_rIter -> first );

   m1_rIter = m1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed multimap is "
        << m1_rIter -> first << "." << endl;
}
```

Output

```
The first element of the reversed multimap m1 is 3.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the first element in the reversed multimap is 2.
```

`multimap::reference`

A type that provides a reference to an element stored in a `multimap`.

C++

```
typedef typename allocator_type::reference reference;
```

Example

C++

```
// multimap_ref.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );

   // Declare and initialize a const_reference &Ref1
   // to the key of the first element
   const int &Ref1 = ( m1.begin( ) -> first );
```

```
    // The following line would cause an error because the
    // non-const_reference can't be used to access the key
    // int &Ref1 = ( m1.begin( ) -> first );

    cout << "The key of first element in the multimap is "
         << Ref1 << "." << endl;

    // Declare and initialize a reference &Ref2
    // to the data value of the first element
    int &Ref2 = ( m1.begin( ) -> second );

    cout << "The data value of first element in the multimap is "
         << Ref2 << "." << endl;

    // The non-const_reference can be used to modify the
    // data value of the first element
    Ref2 = Ref2 + 5;
    cout << "The modified data value of first element is "
         << Ref2 << "." << endl;
}
```

Output

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The modified data value of first element is 15.
```

`multimap::rend`

Returns an iterator that addresses the location succeeding the last element in a reversed `multimap`.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed `multimap` (the location that had preceded the first element in the unreversed `multimap`).

Remarks

`rend` is used with a reversed `multimap` just as end is used with a `multimap`.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the `multimap` object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the `multimap` object can be modified.

`rend` can be used to test whether a reverse iterator has reached the end of its `multimap`.

The value returned by `rend` shouldn't be dereferenced.

Example

C++

```
// multimap_rend.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;

   multimap <int, int> :: iterator m1_Iter;
   multimap <int, int> :: reverse_iterator m1_rIter;
   multimap <int, int> :: const_reverse_iterator m1_crIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );

   m1_rIter = m1.rend( );
   m1_rIter--;
   cout << "The last element of the reversed multimap m1 is "
        << m1_rIter -> first << "." << endl;

   // begin can be used to start an iteration
   // through a multimap in a forward order
   cout << "The multimap is: ";
   for ( m1_Iter = m1.begin( ) ; m1_Iter != m1.end( ); m1_Iter++)
      cout << m1_Iter -> first << " ";
      cout << "." << endl;

   // rbegin can be used to start an iteration
   // through a multimap in a reverse order
   cout << "The reversed multimap is: ";
```

```
    for ( m1_rIter = m1.rbegin( ) ; m1_rIter != m1.rend( ); m1_rIter++)
       cout << m1_rIter -> first << " ";
       cout << "." << endl;

    // A multimap element can be erased by dereferencing to its key
    m1_rIter = --m1.rend( );
    m1.erase ( m1_rIter -> first );

    m1_rIter = m1.rend( );
    m1_rIter--;
    cout << "After the erasure, the last element "
         << "in the reversed multimap is "
         << m1_rIter -> first << "." << endl;
}
```

Output

```
The last element of the reversed multimap m1 is 1.
The multimap is: 1 2 3 .
The reversed multimap is: 3 2 1 .
After the erasure, the last element in the reversed multimap is 2.
```

`multimap::reverse_iterator`

A type that provides a bidirectional iterator that can read or modify an element in a reversed `multimap`.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the `multimap` in reverse.

The `reverse_iterator` defined by `multimap` points to objects of value_type, which are of type `pair<const Key, Type>`. The value of the key is available through the first member pair and the value of the mapped element is available through the second member of the pair.

To dereference a `reverse_iterator` *`rIter`* pointing to an element in a `multimap`, use the `->` operator.

To access the value of the key for the element, use `rIter->first`, which is equivalent to `(*rIter).first`. To access the value of the mapped datum for the element, use `rIter->second`, which is equivalent to `(*rIter).second`.

Example

See the example for rbegin for an example of how to declare and use `reverse_iterator`.

`multimap::size`

Returns the number of elements in the `multimap`.

C++

```
size_type size() const;
```

Return Value

The current length of the `multimap`.

Example

The following example demonstrates the use of the `multimap::size` member function.

C++

```
// multimap_size.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main()
{
    using namespace std;
    multimap<int, int> m1, m2;
    multimap<int, int>::size_type i;
    typedef pair<int, int> Int_Pair;

    m1.insert(Int_Pair(1, 1));
    i = m1.size();
    cout << "The multimap length is " << i << "." << endl;

    m1.insert(Int_Pair(2, 4));
    i = m1.size();
```

```
    cout << "The multimap length is now " << i << "." << endl;
}
```

Output

```
The multimap length is 1.
The multimap length is now 2.
```

`multimap::size_type`

An unsigned integer type that counts the number of elements in a `multimap`.

C++

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for size for an example of how to declare and use `size_type`

`multimap::swap`

Exchanges the elements of two `multimap`s.

C++

```
void swap(
    multimap<Key, Type, Traits, Allocator>& right);
```

Parameters

`right`

The `multimap` providing the elements to be swapped, or the `multimap` whose elements are to be exchanged with those of the `multimap` `left`.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two `multimap`s whose elements are being exchanged.

Example

C++

```
// multimap_swap.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1, m2, m3;
   multimap <int, int>::iterator m1_Iter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m2.insert ( Int_Pair ( 10, 100 ) );
   m2.insert ( Int_Pair ( 20, 200 ) );
   m3.insert ( Int_Pair ( 30, 300 ) );

   cout << "The original multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;

   // This is the member function version of swap
   m1.swap( m2 );

   cout << "After swapping with m2, multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( m1, m3 );

   cout << "After swapping with m3, multimap m1 is:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " " << m1_Iter -> second;
   cout   << "." << endl;
}
```

Output

```
The original multimap m1 is: 10 20 30.
After swapping with m2, multimap m1 is: 100 200.
After swapping with m3, multimap m1 is: 300.
```

`multimap::upper_bound`

Returns an `iterator` to the first element in a `multimap` that with a key that is greater than a specified key.

C++

```
iterator upper_bound(const Key& key);

const_iterator upper_bound(const Key& key) const;
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the `multimap` being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a `multimap` that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the `multimap` if no match is found for the key.

If the return value is assigned to a `const_iterator`, the `multimap` object can't be modified. If the return value is assigned to a `iterator`, the `multimap` object can be modified.

Example

C++

```
// multimap_upper_bound.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   multimap <int, int> m1;
   multimap <int, int> :: const_iterator m1_AcIter, m1_RcIter;
   typedef pair <int, int> Int_Pair;

   m1.insert ( Int_Pair ( 1, 10 ) );
```

```
   m1.insert ( Int_Pair ( 2, 20 ) );
   m1.insert ( Int_Pair ( 3, 30 ) );
   m1.insert ( Int_Pair ( 3, 40 ) );

   m1_RcIter = m1.upper_bound( 1 );
   cout << "The 1st element of multimap m1 with "
        << "a key greater than 1 is: "
        << m1_RcIter -> second << "." << endl;

   m1_RcIter = m1.upper_bound( 2 );
   cout << "The first element of multimap m1 with a key "
        << " greater than 2 is: "
        << m1_RcIter -> second << "." << endl;

   // If no match is found for the key, end( ) is returned
   m1_RcIter = m1.lower_bound( 4 );

   if ( m1_RcIter == m1.end( ) )
      cout << "The multimap m1 doesn't have an element "
           << "with a key of 4." << endl;
   else
      cout << "The element of multimap m1 with a key of 4 is: "
           << m1_RcIter -> second << "." << endl;

   // The element at a specific location in the multimap can be
   // found using a dereferenced iterator addressing the location
   m1_AcIter = m1.begin( );
   m1_RcIter = m1.upper_bound( m1_AcIter -> first );
   cout << "The first element of m1 with a key greater than\n"
        << "that of the initial element of m1 is: "
        << m1_RcIter -> second << "." << endl;
}
```

Output

```
The 1st element of multimap m1 with a key greater than 1 is: 20.
The first element of multimap m1 with a key  greater than 2 is: 30.
The multimap m1 doesn't have an element with a key of 4.
The first element of m1 with a key greater than
that of the initial element of m1 is: 20.
```

`multimap::value_comp`

The member function returns a function object that determines the order of elements in a `multimap` by comparing their key values.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the comparison function object that a `multimap` uses to order its elements.

Remarks

For a `multimap` *`m`*, if two elements `e1(k1, d1)` and `e2(k2, d2)` are objects of type `value_type`, where *`k1`* and *`k2`* are their keys of type `key_type` and *`d1`* and *`d2`* are their data of type `mapped_type`, then `m.value_comp(e1, e2)` is equivalent to `m.key_comp(k1, k2)`.

Example

C++

```
// multimap_value_comp.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;

   multimap <int, int, less<int> > m1;
   multimap <int, int, less<int> >::value_compare vc1 = m1.value_comp( );
   multimap<int,int>::iterator Iter1, Iter2;

   Iter1= m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );
   Iter2= m1.insert ( multimap <int, int> :: value_type ( 2, 5 ) );

   if( vc1( *Iter1, *Iter2 ) == true )
   {
      cout << "The element ( 1,10 ) precedes the element ( 2,5 )."
           << endl;
   }
   else
   {
      cout << "The element ( 1,10 ) does "
           << "not precede the element ( 2,5 )."
           << endl;
   }

   if( vc1( *Iter2, *Iter1 ) == true )
   {
      cout << "The element ( 2,5 ) precedes the element ( 1,10 )."
           << endl;
   }
   else
```

```
   {
      cout << "The element ( 2,5 ) does "
           << "not precede the element ( 1,10 )."
           << endl;
   }
}
```

Output

```
The element ( 1,10 ) precedes the element ( 2,5 ).
The element ( 2,5 ) does not precede the element ( 1,10 ).
```

`multimap::value_type`

A type that represents the type of object stored as an element in a map.

C++

```
typedef pair<const Key, Type> value_type;
```

Example

C++

```
// multimap_value_type.cpp
// compile with: /EHsc
#include <map>
#include <iostream>

int main( )
{
   using namespace std;
   typedef pair <const int, int> cInt2Int;
   multimap <int, int> m1;
   multimap <int, int> :: key_type key1;
   multimap <int, int> :: mapped_type mapped1;
   multimap <int, int> :: value_type value1;
   multimap <int, int> :: iterator pIter;

   // value_type can be used to pass the correct type
   // explicitly to avoid implicit type conversion
   m1.insert ( multimap <int, int> :: value_type ( 1, 10 ) );

   // Compare another way to insert objects into a hash_multimap
   m1.insert ( cInt2Int ( 2, 20 ) );

   // Initializing key1 and mapped1
```

```
    key1 = ( m1.begin( ) -> first );
    mapped1 = ( m1.begin( ) -> second );

    cout << "The key of first element in the multimap is "
         << key1 << "." << endl;

    cout << "The data value of first element in the multimap is "
         << mapped1 << "." << endl;

    // The following line would cause an error because
    // the value_type is not assignable
    // value1 = cInt2Int ( 4, 40 );

    cout  << "The keys of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> first;
    cout << "." << endl;

    cout  << "The values of the mapped elements are:";
    for ( pIter = m1.begin( ) ; pIter != m1.end( ) ; pIter++ )
       cout << " " << pIter -> second;
    cout << "." << endl;
}
```

Output

```
The key of first element in the multimap is 1.
The data value of first element in the multimap is 10.
The keys of the mapped elements are: 1 2.
The values of the mapped elements are: 10 20.
```

See also

Containers

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`value_compare` Class (`<map>`)

Article • 12/06/2021

Provides a function object that can compare the elements of a map by comparing the values of their keys to determine their relative order in the map.

Syntax

C++

```
class value_compare : public binary_function<value_type, value_type, bool>
{
public:
    bool operator()(const value_type& left, const value_type& right) const;
    value_compare(key_compare pred) : comp(pred);
protected:
    key_compare comp;
};
```

Remarks

The comparison criterion provided by `value_compare` between `value_types` of whole elements contained by a map is induced from a comparison between the keys of the respective elements by the auxiliary class construction. The member function operator uses the object `comp` of type `key_compare` stored in the function object provided by `value_compare` to compare the sort-key components of two elements.

For sets and multisets, which are simple containers where the key values are identical to the element values, `value_compare` is equivalent to `key_compare`; for maps and multimaps they are not, as the value of the type `pair` elements is not identical to the value of the element's key.

Example

See example for value_comp for an example of how to declare and use `value_compare`.

Requirements

Header: <map>

Namespace: std

See also

binary_function Struct
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

<memory>

Article • 12/07/2021

Defines a class, an operator, and several templates that help allocate and free objects.

Requirements

Header: <memory>

Namespace: std

Members

Functions

Name	Description
addressof	Gets the true address of an object.
align	Returns a pointer to a range of a given size, based on the provided alignment and starting address.
allocate_shared	Creates a `shared_ptr` to objects that are allocated and constructed for a given type with a specified allocator.
atomic_compare_exchange_strong	
atomic_compare_exchange_weak	
atomic_compare_exchange_strong_explicit	
atomic_compare_exchange_weak_explicit	
atomic_exchange	
atomic_exchange_explicit	
atomic_is_lock_free	
atomic_load	
atomic_load_explicit	
atomic_store	

Name	Description
atomic_store_explicit	
const_pointer_cast	Const cast to `shared_ptr`.
declare_no_pointers	Informs a garbage collector that the characters starting at a specified address and falling within the indicated block size contain no traceable pointers.
declare_reachable	Informs garbage collection that the indicated address is to allocated storage and is reachable.
default_delete	Deletes objects allocated with `operator new`. Suitable for use with `unique_ptr`.
destroy_at	Shorthand `destroy` method.
destroy	Shorthand `destroy` method.
destroy_n	Shorthand `destroy` method.
dynamic_pointer_cast	Dynamic cast to `shared_ptr`.
get_deleter	Get deleter from `shared_ptr`.
get_pointer_safety	Returns the type of pointer safety assumed by any garbage collector.
get_temporary_buffer	Allocates temporary storage for a sequence of elements that does not exceed a specified number of elements.
make_shared	Creates and returns a `shared_ptr` that points to the allocated object constructed from zero or more arguments using the default allocator.
make_unique	Creates and returns a unique_ptr that points to the allocated object constructed from zero or more arguments.
pointer_safety	An enumeration of all the possible return values for `get_pointer_safety`.
return_temporary_buffer	Deallocates the temporary memory that was allocated using the `get_temporary_buffer` template function.
static_pointer_cast	Static cast to `shared_ptr`.
swap	Swap two `shared_ptr` or `weak_ptr` objects.

Name	Description
undeclare_no_pointers	Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size may now contain traceable pointers.
undeclare_reachable	Informs a `garbage_collector` that a specified memory location is not reachable.
uninitialized_copy	Copies objects from a specified input range into an uninitialized destination range.
uninitialized_copy_n	Creates a copy of a specified number of elements from an input iterator. The copies are put in a forward iterator.
uninitialized_default_construct	Shorthand `uninitialized_default_construct` method.
uninitialized_default_construct_n	Shorthand `uninitialized_construct` method.
uninitialized_fill	Copies objects of a specified value into an uninitialized destination range.
uninitialized_fill_n	Copies objects of a specified value into specified number of elements an uninitialized destination range.
uninitialized_move	Shorthand `uninitialized_move` method.
uninitialized_move_n	Shorthand `uninitialized_move` method.
uninitialized_value_construct	Shorthand `uninitialized_value_construct` method.
uninitialized_value_construct_n	Shorthand `uninitialized_value_construct` method.
uses_allocator_v	

Operators

Name	Description
operator!=	Tests for inequality between allocator objects of a specified class.
operator==	Tests for equality between allocator objects of a specified class.
operator>=	Tests for one allocator object being greater than or equal to a second allocator object, of a specified class.
operator<	Tests for one object being less than a second object of a specified class.

Name	Description
operator<=	Tests for one object being less than or equal to a second object of a specified class.
operator>	Tests for one object being greater than a second object of a specified class.
operator<<	`shared_ptr` inserter.

Classes

Name	Description
allocator	The class template describes an object that manages storage allocation and freeing for arrays of objects of type **Type**.
allocator_traits	Describes an object that determines all the information that is needed by an allocator-enabled container.
auto_ptr	The class template describes an object that stores a pointer to an allocated object of type **Type** * that ensures the object to which it points gets deleted when its enclosing auto_ptr gets destroyed.
bad_weak_ptr	Reports bad weak_ptr exception.
enabled_shared_from_this	Helps generate a `shared_ptr`.
pointer_traits	Supplies information that is needed by an object of type `allocator_traits` to describe an allocator with pointer type `Ptr`.
raw_storage_iterator	An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.
shared_ptr	Wraps a reference-counted smart pointer around a dynamically allocated object.
unique_ptr	Stores a pointer to an owned object. The pointer is owned by no other `unique_ptr`. The `unique_ptr` is destroyed when the owner is destroyed.
weak_ptr	Wraps a weakly linked pointer.

Structures

Name	Description
allocator_arg_t	
default_delete	

Name	Description
hash	Provides overloads specialized for `unique_ptr` and `shared_ptr`.
owner_less	Allows ownership-based mixed comparisons of shared and weak pointers.
uses_allocator	

Specializations

Name	Description
allocator<void>	A specialization of the class template allocator to type `void`, defining only the member types that make sense in this specialized context.

See also

Header Files Reference
Thread Safety in the C++ Standard Library

`<memory>` functions

Article • 11/07/2022

`addressof`

Gets the true address of an object.

C++

```
template <class T>
T* addressof(
    T& value) noexcept;    // before C++17

template <class T>
constexpr T* addressof(
    T& value) noexcept;    // C++17

template <class T>
const T* addressof(
    const T&& value) = delete;   // C++17
```

Parameters

`value`

The object or function for which to obtain the true address.

Return Value

The actual address of the object or function referenced by *`value`*, even if an overloaded `operator&()` exists.

Remarks

`align`

Fits storage of the given size, aligned by the given alignment specification, into the first possible address of the given storage.

C++

```
void* align(
    size_t alignment, // input
    size_t size,      // input
    void*& ptr,       // input/output
    size_t& space     // input/output
);
```

Parameters

alignment

The alignment bound to attempt.

size

The size in bytes for the aligned storage.

ptr

The starting address of the available contiguous storage pool to use. This parameter is also an output parameter, and is set to contain the new starting address if the alignment is successful. If `align()` is unsuccessful, this parameter isn't modified.

space

The total space available to `align()` to use in creating the aligned storage. This parameter is also an output parameter, and contains the adjusted space left in the storage buffer after the aligned storage and any associated overhead is subtracted.

If `align()` is unsuccessful, this parameter isn't modified.

Return Value

A `NULL` pointer if the requested aligned buffer wouldn't fit into the available space; otherwise, the new value of *ptr*.

Remarks

The modified *ptr* and *space* parameters enable you to call `align()` repeatedly on the same buffer, possibly with different values for *alignment* and *size*. The following code snippet shows one use of `align()`.

C++

```
#include <type_traits> // std::alignment_of()
#include <memory>
```

```
//...
char buffer[256]; // for simplicity
size_t alignment = std::alignment_of<int>::value;
void * ptr = buffer;
std::size_t space = sizeof(buffer); // Be sure this results in the true size
of your buffer

while (std::align(alignment, sizeof(MyObj), ptr, space)) {
    // You now have storage the size of MyObj, starting at ptr, aligned on
    // int boundary. Use it here if you like, or save off the starting
address
    // contained in ptr for later use.
    // ...
    // Last, move starting pointer and decrease available space before
    // the while loop restarts.
    ptr = reinterpret_cast<char*>(ptr) + sizeof(MyObj);
    space -= sizeof(MyObj);
}
// At this point, align() has returned a null pointer, signaling it is not
// possible to allow more aligned storage in this buffer.
```

`allocate_shared`

Creates a shared_ptr to objects that are allocated and constructed for a given type by using a specified allocator. Returns the `shared_ptr`.

C++

```
template <class T, class Allocator, class... Args>
shared_ptr<T> allocate_shared(
    Allocator alloc,
    Args&&... args);
```

Parameters

`alloc`

The allocator used to create objects.

`args`

The zero or more arguments that become the objects.

Remarks

The function creates the object `shared_ptr<T>`, a pointer to `T(args...)` as allocated and constructed by *`alloc`*.

atomic_compare_exchange_strong

C++

```
template<class T>
bool atomic_compare_exchange_strong(
    shared_ptr<T>* u,
    shared_ptr<T>* v,
    shared_ptr<T> w);
```

atomic_compare_exchange_weak

C++

```
template<class T>
bool atomic_compare_exchange_weak(
    shared_ptr<T>* u,
    shared_ptr<T>* v,
    shared_ptr<T> w);
```

atomic_compare_exchange_strong_explicit

C++

```
template<class T>
bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* u,
    shared_ptr<T>* v,
    shared_ptr<T> w,
    memory_order success,
    memory_order failure);
```

atomic_compare_exchange_weak_explicit

C++

```
template<class T>
bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* u,
    shared_ptr<T>* v,
    shared_ptr<T> w,
    memory_order success,
    memory_order failure);
```

atomic_exchange

C++

```
template<class T>
shared_ptr<T> atomic_exchange(
    shared_ptr<T>* u,
    shared_ptr<T> r);
```

atomic_exchange_explicit

C++

```
template<class T>
shared_ptr<T> atomic_exchange_explicit(
    shared_ptr<T>* u,
    shared_ptr<T> r,
    memory_order mo);
```

atomic_is_lock_free

C++

```
template<class T>
bool atomic_is_lock_free(
    const shared_ptr<T>* u);
```

atomic_load

C++

```
template<class T>
shared_ptr<T> atomic_load(
    const shared_ptr<T>* u);
```

atomic_load_explicit

C++

```
template<class T>
shared_ptr<T> atomic_load_explicit(
```

```
    const shared_ptr<T>* u,
    memory_order mo);
```

atomic_store

C++

```
template<class T>
void atomic_store(
    shared_ptr<T>* u,
    shared_ptr<T> r);
```

atomic_store_explicit

C++

```
template<class T>
void atomic_store_explicit(
    shared_ptr<T>* u,
    shared_ptr<T> r,
    memory_order mo);
```

const_pointer_cast

Const cast to shared_ptr.

C++

```
template <class T, class Other>
shared_ptr<T> const_pointer_cast(
    const shared_ptr<Other>& sp) noexcept;

template <class T, class Other>
shared_ptr<T> const_pointer_cast(
    shared_ptr<Other>&& sp) noexcept;
```

Parameters

T

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

sp

The argument shared pointer.

Remarks

The template function returns an empty `shared_ptr` object if `const_cast<T*>(sp.get())` returns a null pointer; otherwise it returns a `shared_ptr<T>` object that owns the resource that is owned by *sp*. The expression `const_cast<T*>(sp.get())` must be valid.

Example

C++

```
// std__memory__const_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int);
    std::shared_ptr<const int> sp1 =
        std::const_pointer_cast<const int>(sp0);

    *sp0 = 3;
    std::cout << "sp1 == " << *sp1 << std::endl;

    return (0);
}
```

Output

```
sp1 == 3
```

declare_no_pointers

Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size contains no traceable pointers.

C++

```
void declare_no_pointers(
    char* ptr,
    size_t size);
```

Parameters

ptr
Address of first character that no longer contains traceable pointers.

size
Size of block that starts at *ptr* that contains no traceable pointers.

Remarks

The function informs any garbage collector that the addresses in the range `[ptr, ptr + size)` no longer contain traceable pointers. (Any pointers to allocated storage must not be dereferenced unless made reachable.)

declare_reachable

Informs garbage collection that the indicated address is to allocated storage and is reachable.

C++

```
void declare_reachable(
    void* ptr);
```

Parameters

ptr
A pointer to a reachable, allocated, valid storage area.

Remarks

If *ptr* is not null, the function informs any garbage collector that *ptr* is now reachable, that is, it points to valid allocated storage.

default_delete

Deletes objects allocated with `operator new`. Suitable for use with unique_ptr.

```
C++
```

```
struct default_delete
{
    constexpr default_delete() noexcept = default;

    template <class Other, class = typename enable_if<is_convertible<Other*,
T*>::value, void>::type>>
    default_delete(const default_delete<Other>&) noexcept;

    void operator()(T* ptr) const noexcept;
};
```

Parameters

ptr

Pointer to the object to delete.

Other

The type of elements in the array to be deleted.

Remarks

The class template describes a deleter that deletes scalar objects allocated with `operator new`, suitable for use with class template `unique_ptr`. It also has the explicit specialization `default_delete<T[]>`.

destroy_at

```
C++
```

```
template <class T>
void destroy_at(
    T* location);
```

Same as `location->~T()`.

destroy

```
C++
```

```
template <class ForwardIterator>
void destroy(
    ForwardIterator first,
    ForwardIterator last);
```

Same as:

C++

```
for (; first != last; ++first)
    destroy_at(addressof(*first));
```

destroy_n

C++

```
template <class ForwardIterator, class Size>
ForwardIterator destroy_n(
    ForwardIterator first,
    Size count);
```

Same as:

C++

```
for (; count > 0; (void)++first, --count)
    destroy_at(addressof(*first));
return first;
```

dynamic_pointer_cast

Dynamic cast to shared_ptr.

C++

```
template <class T, class Other>
shared_ptr<T> dynamic_pointer_cast(
    const shared_ptr<Other>& sp) noexcept;

template <class T, class Other>
shared_ptr<T> dynamic_pointer_cast(
    shared_ptr<Other>&& sp) noexcept;
```

Parameters

T

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

sp

The argument shared pointer.

Remarks

The template function returns an empty `shared_ptr` object if `dynamic_cast<T*>(sp.get())` returns a null pointer; otherwise it returns a `shared_ptr<T>` object that owns the resource that is owned by *sp*. The expression `dynamic_cast<T*>(sp.get())` must be valid.

Example

C++

```
// std__memory__dynamic_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct base
{
    virtual ~base() {}
    int value;
};

struct derived
    : public base
{
};

int main()
{
    std::shared_ptr<base> sp0(new derived);
    std::shared_ptr<derived> sp1 =
        std::dynamic_pointer_cast<derived>(sp0);

    sp0->value = 3;
    std::cout << "sp1->value == " << sp1->value << std::endl;
```

```
    return (0);
}
```

Output

```
sp1->value == 3
```

`get_deleter`

Get the deleter from a shared_ptr.

C++

```
template <class Deleter, class T>
Deleter* get_deleter(
    const shared_ptr<T>& sp) noexcept;
```

Parameters

`Deleter`

The type of the deleter.

`T`

The type controlled by the shared pointer.

`sp`

The shared pointer.

Remarks

The template function returns a pointer to the deleter of type *`Deleter`* that belongs to the `shared_ptr` object *`sp`*. If *`sp`* has no deleter, or if its deleter is not of type *`Deleter`*, the function returns 0.

Example

C++

```
// std__memory__get_deleter.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
```

```
struct base
{
    int value;
};

struct deleter
{
    void operator()(base *pb)
    {
        delete pb;
    }
};

int main()
{
    std::shared_ptr<base> sp0(new base);

    sp0->value = 3;
    std::cout << "get_deleter(sp0) != 0 == " << std::boolalpha
        << (std::get_deleter<deleter>(sp0) != 0) << std::endl;

    std::shared_ptr<base> sp1(new base, deleter());

    sp0->value = 3;
    std::cout << "get_deleter(sp1) != 0 == " << std::boolalpha
        << (std::get_deleter<deleter>(sp1) != 0) << std::endl;

    return (0);
}
```

Output

```
get_deleter(sp0) != 0 == false
get_deleter(sp1) != 0 == true
```

`get_pointer_safety`

Returns the type of pointer safety assumed by any garbage collector.

C++

```
pointer_safety get_pointer_safety() noexcept;
```

Remarks

The function returns the type of pointer safety assumed by any automatic garbage collector.

`get_temporary_buffer`

Allocates temporary storage for a sequence of elements that doesn't exceed a specified number of elements.

```
C++
```

```
template <class T>
pair<T *, ptrdiff_t> get_temporary_buffer(
    ptrdiff_t count);
```

Parameters

count
The maximum number of elements requested for which memory is to be allocated.

Return Value

A `pair` whose first component is a pointer to the memory that was allocated, and whose second component gives the size of the buffer, indicating the largest number of elements that it could store.

Remarks

The function makes a request for memory and it may not succeed. If no buffer is allocated, then the function returns a pair, with the second component equal to zero and the first component equal to the null pointer.

Only use this function for memory that is temporary.

Example

```
C++
```

```
// memory_get_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
```

```
using namespace std;

int main( )
{
    // Create an array of ints
    int intArray [] = { 10, 20, 30, 40, 100, 200, 300, 1000, 2000 };
    int count = sizeof ( intArray ) / sizeof ( int );
    cout << "The number of integers in the array is: "
        << count << "." << endl;

    pair<int *, ptrdiff_t> resultPair;
    resultPair = get_temporary_buffer<int>( count );

    cout << "The number of elements that the allocated memory\n"
        << "could store is given by: resultPair.second = "
        << resultPair.second << "." << endl;
}
```

Output

```
The number of integers in the array is: 9.
The number of elements that the allocated memory
could store is given by: resultPair.second = 9.
```

`make_shared`

Creates and returns a shared_ptr that points to the allocated objects that are constructed from zero or more arguments by using the default allocator. Allocates and constructs both an object of the specified type and a `shared_ptr` to manage shared ownership of the object, and returns the `shared_ptr`.

C++

```
template <class T, class... Args>
shared_ptr<T> make_shared(
    Args&&... args);
```

Parameters

`args`

Zero or more constructor arguments. The function infers which constructor overload to invoke based on the arguments that are provided.

Remarks

Use `make_shared` as a simple and more efficient way to create an object and a `shared_ptr` to manage shared access to the object at the same time. Semantically, these two statements are equivalent:

```
C++
```

```
auto sp = std::shared_ptr<Example>(new Example(argument));
auto msp = std::make_shared<Example>(argument);
```

However, the first statement makes two allocations, and if the allocation of the `shared_ptr` fails after the allocation of the `Example` object has succeeded, then the unnamed `Example` object is leaked. The statement that uses `make_shared` is simpler because there's only one function call involved. It's more efficient because the library can make a single allocation for both the object and the smart pointer. This function is both faster and leads to less memory fragmentation, and there's no chance of an exception on one allocation but not the other. Performance is improved by better locality for code that references the object and updates the reference counts in the smart pointer.

Consider using make_unique if you don't need shared access to the object. Use allocate_shared if you need to specify a custom allocator for the object. You can't use `make_shared` if your object requires a custom deleter, because there's no way to pass the deleter as an argument.

The following example shows how to create shared pointers to a type by invoking specific constructor overloads.

Example

```
C++
```

```
// stl_make_shared.cpp
// Compile by using: cl /W4 /EHsc stl_make_shared.cpp
#include <iostream>
#include <string>
#include <memory>
#include <vector>

class Song {
public:
    std::wstring title_;
    std::wstring artist_;

    Song(std::wstring title, std::wstring artist) : title_(title),
artist_(artist) {}
```

```
    Song(std::wstring title) : title_(title), artist_(L"Unknown") {}
};

void CreateSharedPointers()
{
    // Okay, but less efficient to have separate allocations for
    // Song object and shared_ptr control block.
    auto song = new Song(L"Ode to Joy", L"Beethoven");
    std::shared_ptr<Song> sp0(song);

    // Use make_shared function when possible. Memory for control block
    // and Song object are allocated in the same call:
    auto sp1 = std::make_shared<Song>(L"Yesterday", L"The Beatles");
    auto sp2 = std::make_shared<Song>(L"Blackbird", L"The Beatles");

    // make_shared infers which constructor to use based on the arguments.
    auto sp3 = std::make_shared<Song>(L"Greensleeves");

    // The playlist vector makes copies of the shared_ptr pointers.
    std::vector<std::shared_ptr<Song>> playlist;
    playlist.push_back(sp0);
    playlist.push_back(sp1);
    playlist.push_back(sp2);
    playlist.push_back(sp3);
    playlist.push_back(sp1);
    playlist.push_back(sp2);
    for (auto&& sp : playlist)
    {
        std::wcout << L"Playing " << sp->title_ <<
            L" by " << sp->artist_ << L", use count: " <<
            sp.use_count() << std::endl;
    }
}

int main()
{
    CreateSharedPointers();
}
```

The example produces this output:

Output

```
Playing Ode to Joy by Beethoven, use count: 2
Playing Yesterday by The Beatles, use count: 3
Playing Blackbird by The Beatles, use count: 3
Playing Greensleeves by Unknown, use count: 2
Playing Yesterday by The Beatles, use count: 3
Playing Blackbird by The Beatles, use count: 3
```

make_unique

Creates and returns a unique_ptr to an object of the specified type, which is constructed by using the specified arguments.

```
C++
```

```
// make_unique<T>
template <class T, class... Args>
unique_ptr<T> make_unique(Args&&... args);

// make_unique<T[]>
template <class T>
unique_ptr<T> make_unique(size_t size);

// make_unique<T[N]> disallowed
template <class T, class... Args>
/* unspecified */ make_unique(Args&&...) = delete;
```

Parameters

`T`

The type of the object that the `unique_ptr` will point to.

`Args`

The types of the constructor arguments specified by `args`.

`args`

The arguments to be passed to the constructor of the object of type `T`.

`elements`

An array of elements of type `T`.

`size`

The number of elements to allocate space for in the new array.

Remarks

The first overload is used for single objects. The second overload is invoked for arrays. The third overload prevents you from specifying an array size in the type argument (`make_unique<T[N]>`); this construction isn't supported by the current standard. When you use `make_unique` to create a `unique_ptr` to an array, you have to initialize the array elements separately. Rather than using this overload, perhaps a better choice is to use a std::vector.

Because `make_unique` is carefully implemented for exception safety, we recommend that you use `make_unique` instead of directly calling `unique_ptr` constructors.

Example

The following example shows how to use `make_unique`. For more examples, see How to: Create and Use unique_ptr Instances.

C++

```
class Animal
{
private:
    std::wstring genus;
    std::wstring species;
    int age;
    double weight;
public:
    Animal(const wstring&, const wstring&, int, double){/*...*/ }
    Animal(){}
};

void MakeAnimals()
{
    // Use the Animal default constructor.
    unique_ptr<Animal> p1 = make_unique<Animal>();

    // Use the constructor that matches these arguments
    auto p2 = make_unique<Animal>(L"Felis", L"Catus", 12, 16.5);

    // Create a unique_ptr to an array of 5 Animals
    unique_ptr<Animal[]> p3 = make_unique<Animal[]>(5);

    // Initialize the elements
    p3[0] = Animal(L"Rattus", L"norvegicus", 3, 2.1);
    p3[1] = Animal(L"Corynorhinus", L"townsendii", 4, 1.08);

    // auto p4 = p2; //C2280

    vector<unique_ptr<Animal>> vec;
    // vec.push_back(p2); //C2280
    // vector<unique_ptr<Animal>> vec2 = vec; // C2280

    // OK. p2 no longer points to anything
    vec.push_back(std::move(p2));

    // unique_ptr overloads operator bool
    wcout << boolalpha << (p2 == false) << endl; // Prints "true"

    // OK but now you have two pointers to the same memory location
    Animal* pAnimal = p2.get();
```

```
    // OK. p2 no longer points to anything
    Animal* p5 = p2.release();
}
```

When you see error C2280 in connection with a `unique_ptr`, it is almost certainly because you are attempting to invoke its copy constructor, which is a deleted function.

`owner_less`

Allows ownership-based mixed comparisons of shared and weak pointers. Returns **`true`** if the left parameter is ordered before right parameter by the member function `owner_before`.

C++

```
template <class T>
    struct owner_less; // not defined

template <class T>
struct owner_less<shared_ptr<T>>
{
    bool operator()(
        const shared_ptr<T>& left,
        const shared_ptr<T>& right) const noexcept;

    bool operator()(
        const shared_ptr<T>& left,
        const weak_ptr<T>& right) const noexcept;

    bool operator()(
        const weak_ptr<T>& left,
        const shared_ptr<T>& right) const noexcept;
};

template <class T>
struct owner_less<weak_ptr<T>>
    bool operator()(
        const weak_ptr<T>& left,
        const weak_ptr<T>& right) const noexcept;

    bool operator()(
        const weak_ptr<T>& left,
        const shared_ptr<T>& right) const noexcept;

    bool operator()(
        const shared_ptr<T>& left,
        const weak_ptr<T>& right) const noexcept;
};
```

```
template<> struct owner_less<void>
{
    template<class T, class U>
    bool operator()(
        const shared_ptr<T>& left,
        const shared_ptr<U>& right) const noexcept;

    template<class T, class U>
    bool operator()(
        const shared_ptr<T>& left,
        const weak_ptr<U>& right) const noexcept;

    template<class T, class U>
    bool operator()(
        const weak_ptr<T>& left,
        const shared_ptr<U>& right) const noexcept;

    template<class T, class U>
    bool operator()(
        const weak_ptr<T>& left,
        const weak_ptr<U>& right) const noexcept;
};
```

Parameters

left

A shared or weak pointer.

right

A shared or weak pointer.

Remarks

The class templates define all their member operators as returning `left.owner_before(right)`.

`reinterpret_pointer_cast`

Creates a new `shared_ptr` from an existing shared pointer by using a cast.

C++

```
template<class T, class U>
shared_ptr<T> reinterpret_pointer_cast(
    const shared_ptr<U>& ptr) noexcept;

template<class T, class U>
```

```
shared_ptr<T> reinterpret_pointer_cast(
    shared_ptr<U>&& ptr) noexcept;
```

Parameters

ptr
An reference to a `shared_ptr<U>`.

Remarks

If *ptr* is empty, the new `shared_ptr` is also empty, otherwise it shares ownership with *ptr*. The new shared pointer is the result of evaluating `reinterpret_cast<Y*>(ptr.get())`, where `Y` is `typename std::shared_ptr<T>::element_type`. The behavior is undefined if `reinterpret_cast<T*>((U*)nullptr)` is not well-formed.

The template function that takes an lvalue reference is new in C++17. The template function that takes an rvalue reference is new in C++20.

`return_temporary_buffer`

Deallocates the temporary memory that was allocated using the `get_temporary_buffer` template function.

C++

```
template <class T>
void return_temporary_buffer(
    T* buffer);
```

Parameters

buffer
A pointer to the memory to be deallocated.

Remarks

Only use this function for memory that is temporary.

Example

C++

```
// memory_ret_temp_buf.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

int main( )
{
    // Create an array of ints
    int intArray [] = { 10, 20, 30, 40, 100, 200, 300 };
    int count = sizeof ( intArray ) / sizeof ( int );
    cout << "The number of integers in the array is: "
         << count << "." << endl;

    pair<int *, ptrdiff_t> resultPair;
    resultPair = get_temporary_buffer<int>( count );

    cout << "The number of elements that the allocated memory\n"
         << " could store is given by: resultPair.second = "
         << resultPair.second << "." << endl;

    int* tempBuffer = resultPair.first;

    // Deallocates memory allocated with get_temporary_buffer
    return_temporary_buffer( tempBuffer );
}
```

Output

```
The number of integers in the array is: 7.
The number of elements that the allocated memory
could store is given by: resultPair.second = 7.
```

static_pointer_cast

Static cast to shared_ptr.

C++

```
template <class T, class Other>
shared_ptr<T> static_pointer_cast(
    const shared_ptr<Other>& sp) noexcept;

template <class T, class Other>
shared_ptr<T> static_pointer_cast(
    shared_ptr<Other>&& sp) noexcept;
```

Parameters

T

The type controlled by the returned shared pointer.

Other

The type controlled by the argument shared pointer.

sp

The argument shared pointer.

Remarks

The template function returns an empty `shared_ptr` object if *sp* is an empty `shared_ptr` object; otherwise it returns a `shared_ptr<T>` object that owns the resource that is owned by *sp*. The expression `static_cast<T*>(sp.get())` must be valid.

Example

C++

```
// std__memory__static_pointer_cast.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct base
{
    int value;
};

struct derived
    : public base
{
};

int main()
{
    std::shared_ptr<base> sp0(new derived);
    std::shared_ptr<derived> sp1 =
        std::static_pointer_cast<derived>(sp0);

    sp0->value = 3;
    std::cout << "sp1->value == " << sp1->value << std::endl;

    return (0);
}
```

Output

```
sp1->value == 3
```

swap

Swap two shared_ptr, unique_ptr, or weak_ptr objects.

C++

```
template <class T>
void swap(
    shared_ptr<T>& left,
    shared_ptr<T>& right) noexcept;

template <class T, class Deleter>
void swap(
    unique_ptr<T, Deleter>& left,
    unique_ptr<T, Deleter>& right) noexcept;

template <class T>
void swap(
    weak_ptr<T>& left,
    weak_ptr<T>& right) noexcept;
```

Parameters

T

The type controlled by the argument pointer.

Deleter

The deleter of the unique pointer type.

left

The left pointer.

right

The right pointer.

Remarks

The template functions call `left.swap(right)`.

Example

C++

```
// std__memory__swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}
```

Output

```
*sp1 == 5
*sp1 == 10
*sp1 == 5

*wp1 == 5
*wp1 == 10
*wp1 == 5
```

`undeclare_no_pointers`

Informs a garbage collector that the characters in the memory block defined by a base address pointer and block size may now contain traceable pointers.

C++

```
void undeclare_no_pointers(
    char* ptr,
    size_t size);
```

Parameters

ptr

A pointer to the memory address previously marked using declare_no_pointers.

size

The number of bytes in the memory range. This value must equal the number used in the `declare_no_pointers` call.

Remarks

The function informs any garbage collector that the range of addresses `[ptr, ptr + size)` may now contain traceable pointers.

`undeclare_reachable`

Revokes a declaration of reachability for a specified memory location.

C++

```
template <class T>
T *undeclare_reachable(
    T* ptr);
```

Parameters

ptr

A pointer to the memory address previously marked using declare_reachable.

Remarks

If *ptr* is not **nullptr**, the function informs any garbage collector that *ptr* is no longer reachable. It returns a safely derived pointer that compares equal to *ptr*.

`uninitialized_copy`

Copies objects from a specified source range into an uninitialized destination range.

```
C++
```

```
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(
    InputIterator first,
    InputIterator last,
    ForwardIterator dest);

template <class ExecutionPolicy, class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(
    ExecutionPolicy&& policy,
    InputIterator first,
    InputIterator last,
    ForwardIterator dest);
```

Parameters

`policy`

The execution policy to use.

`first`

An input iterator addressing the first element in the source range.

`last`

An input iterator addressing the last element in the source range.

`dest`

A forward iterator addressing the first element in the destination range.

Return Value

A forward iterator addressing the first position beyond the destination range, unless the source range was empty.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
```

C++

```
while (first != last)
{
    new (static_cast<void*>(&* dest++))
        typename iterator_traits<InputIterator>::value_type(*first++);
}
return dest;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

The overload with an execution policy is new in C++17.

Example

C++

```
// memory_uninit_copy.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

class Integer
{
public:
    Integer(int x) : value(x) {}
    int get() { return value; }
private:
    int value;
};

int main()
{
    int Array[] = { 10, 20, 30, 40 };
    const int N = sizeof(Array) / sizeof(int);

    cout << "The initialized Array contains " << N << " elements: ";
    for (int i = 0; i < N; i++)
    {
        cout << " " << Array[i];
    }
    cout << endl;

    Integer* ArrayPtr = (Integer*)malloc(N * sizeof(int));
    Integer* LArrayPtr = uninitialized_copy(
        Array, Array + N, ArrayPtr);  // C4996

    cout << "Address of position after the last element in the array is: "
```

```
        << &Array[0] + N << endl;
    cout << "The iterator returned by uninitialized_copy addresses: "
        << (void*)LArrayPtr << endl;
    cout << "The address just beyond the last copied element is: "
        << (void*)(ArrayPtr + N) << endl;

    if ((&Array[0] + N) == (void*)LArrayPtr)
        cout << "The return value is an iterator "
        << "pointing just beyond the original array." << endl;
    else
        cout << "The return value is an iterator "
        << "not pointing just beyond the original array." << endl;

    if ((void*)LArrayPtr == (void*)(ArrayPtr + N))
        cout << "The return value is an iterator "
        << "pointing just beyond the copied array." << endl;
    else
        cout << "The return value is an iterator "
        << "not pointing just beyond the copied array." << endl;

    free(ArrayPtr);

    cout << "Note that the exact addresses returned will vary\n"
        << "with the memory allocation in individual computers."
        << endl;
}
```

`uninitialized_copy_n`

Creates a copy of a specified number of elements from an input iterator. The copies are put in a forward iterator.

C++

```
template <class InputIterator, class Size, class ForwardIterator>
ForwardIterator uninitialized_copy_n(
    InputIterator first,
    Size count,
    ForwardIterator dest);

template <class ExecutionPolicy, class InputIterator, class Size, class
ForwardIterator>
ForwardIterator uninitialized_copy_n(
    ExecutionPolicy&& policy,
    InputIterator first,
    Size count,
    ForwardIterator dest);
```

Parameters

policy

The execution policy to use.

first

An input iterator that refers to the object to copy.

count

A signed or unsigned integer type specifying the number of times to copy the object.

dest

A forward iterator that refers to where the new copies go.

Return Value

A forward iterator that addresses the first position beyond the destination. If the source range was empty, the iterator addresses *first*.

Remarks

The template function effectively executes the following code:

C++

```
    for (; 0 < count; --count)
        new (static_cast<void*>(&* dest++))
            typename iterator_traits<InputIterator>::value_type(*first++);
    return dest;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

The overload with an execution policy is new in C++17.

`uninitialized_default_construct`

Default constructs objects of the iterators' `value_type` in the specified range.

C++

```
template <class ForwardIterator>
void uninitialized_default_construct(
    ForwardIterator first,
    ForwardIterator last);
```

```
template <class ExecutionPolicy, class ForwardIterator>
void uninitialized_default_construct(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    ForwardIterator last);
```

Parameters

policy

The execution policy to use.

first

An iterator addressing the first element in the range to construct.

last

An iterator addressing one past the last element in the range to construct.

Remarks

The version without an execution policy is effectively the same as:

C++

```
for (; first != last; ++first)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type;
```

If an exception is thrown, previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified *policy*.

These functions are new in C++17.

`uninitialized_default_construct_n`

Default constructs a specified number of objects of the iterator's `value_type`, starting at the specified location.

C++

```
template <class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(
```

```
    ForwardIterator first,
    Size count);

template <class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    Size count);
```

Parameters

policy

The execution policy to use.

first

An iterator addressing the first element in the destination range to construct.

count

The count of elements in the destination range to construct.

Return Value

A forward iterator addressing the first position beyond the destination range, unless the source range was empty.

Remarks

The version without an execution policy is effectively the same as:

C++

```
for (; count>0; (void)++first, --count)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type;
return first;
```

If an exception is thrown, previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified *policy*.

These functions are new in C++17.

`uninitialized_fill`

Copies objects of a specified value into an uninitialized destination range.

```
C++
```

```
template <class ForwardIterator, class T>
void uninitialized_fill(
    ForwardIterator first,
    ForwardIterator last,
    const T& value);

template <class ExecutionPolicy, class ForwardIterator, class T>
void uninitialized_fill(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    ForwardIterator last,
    const T& value);
```

Parameters

`policy`

The execution policy to use.

`first`

A forward iterator addressing the first element in the destination range to initialize.

`last`

A forward iterator addressing the last element in the destination range to initialize.

`value`

The value to be used to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

```
C++
```

```
while (first != last)
    new (static_cast<void*>(&* first ++))
        typename iterator_traits<ForwardIterator>::value_type (value);
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

The overload with an execution policy is new in C++17.

Example

C++

```
// memory_uninit_fill.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

class Integer
{
public:
    // No default constructor
    Integer( int x ) : value( x ) {}
    int get() { return value; }
private:
    int value;
};

int main()
{
    const int N = 10;
    Integer value ( 25 );
    Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
    uninitialized_fill( Array, Array + N, value );
    cout << "The initialized Array contains: ";
    for ( int i = 0; i < N; i++ )
        {
            cout << Array[ i ].get() << " ";
        }
    cout << endl;
}
```

Output

```
The initialized Array contains: 25 25 25 25 25 25 25 25 25 25
```

`uninitialized_fill_n`

Copies objects of a specified value into the specified number of elements of an uninitialized destination range.

C++

```
template <class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(
    ForwardIterator first,
    Size count,
    const T& value);

template <class ExecutionPolicy, class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    Size count,
    const T& value);
```

Parameters

policy

The execution policy to use.

first

A forward iterator addressing the first element in the destination range to initialize.

count

The number of elements to initialize.

value

The value to use to initialize the destination range.

Remarks

This algorithm allows the decoupling of memory allocation from object construction.

The template function effectively executes:

C++

```
while (0 < count--)
    new (static_cast<void*>(&* first++))
        typename iterator_traits<ForwardIterator>::value_type(value);
return first;
```

unless the code throws an exception. In that case, all constructed objects are destroyed and the exception is rethrown.

The overload with an execution policy is new in C++17.

Example

C++

```
// memory_uninit_fill_n.cpp
// compile with: /EHsc /W3
#include <memory>
#include <iostream>

using namespace std;

class Integer
{
public:
    // No default constructor
    Integer( int x ) : value( x ) {}
    int get() { return value; }
private:
    int value;
};

int main()
{
    const int N = 10;
    Integer value( 60 );
    Integer* Array = ( Integer* ) malloc( N * sizeof( int ) );
    uninitialized_fill_n( Array, N, value );  // C4996
    cout << "The uninitialized Array contains: ";
    for ( int i = 0; i < N; i++ )
        cout << Array[ i ].get() <<  " ";
}
```

`uninitialized_move`

Moves elements from a source range to an uninitialized destination memory area.

C++

```
template <class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_move(
    InputIterator first,
    InputIterator last,
    ForwardIterator dest);

template <class ExecutionPolicy, class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_move(
    ExecutionPolicy&& policy,
    InputIterator first,
    InputIterator last,
    ForwardIterator dest);
```

Parameters

policy

The execution policy to use.

first

An input iterator addressing the first element in the source range to move.

last

An input iterator addressing one past the last element in the source range to move.

dest

The beginning of the destination range.

Remarks

The version without an execution policy is effectively the same as:

```
C++
```

```
for (; first != last; (void)++dest, ++first)
    ::new (static_cast<void*>(addressof(*dest)))
        typename
iterator_traits<ForwardIterator>::value_type(std::move(*first));
return dest;
```

If an exception is thrown, some objects in the source range might be left in a valid but unspecified state. Previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified *policy*.

These functions are new in C++17.

`uninitialized_move_n`

Moves a specified number of elements from a source range to an uninitialized destination memory area.

```
C++
```

```
template <class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator> uninitialized_move_n(
```

```
    InputIterator first,
    Size count,
    ForwardIterator dest);

template <class ExecutionPolicy, class InputIterator, class Size, class
ForwardIterator>
pair<InputIterator, ForwardIterator> uninitialized_move_n(
    ExecutionPolicy&& policy,
    InputIterator first,
    Size count,
    ForwardIterator dest);
```

Parameters

policy

The execution policy to use.

first

An input iterator addressing the first element in the source range to move.

count

The count of elements in the source range to move.

dest

The beginning of the destination range.

Remarks

The version without an execution policy is effectively the same as:

C++

```
for (; count > 0; ++dest, (void) ++first, --count)
    ::new (static_cast<void*>(addressof(*dest)))
        typename
iterator_traits<ForwardIterator>::value_type(std::move(*first));
return {first, dest};
```

If an exception is thrown, some objects in the source range might be left in a valid but unspecified state. Previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified *policy*.

These functions are new in C++17.

`uninitialized_value_construct`

Constructs objects of the iterators' `value_type` by value initialization, in the specified range.

```
C++
```

```
template <class ForwardIterator>
void uninitialized_value_construct(
    ForwardIterator first,
    ForwardIterator last);

template <class ExecutionPolicy, class ForwardIterator>
void uninitialized_value_construct(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    ForwardIterator last);
```

Parameters

`policy`
The execution policy to use.

`first`
An iterator addressing the first element in the range to value construct.

`last`
An iterator addressing one past the last element in the range to value construct.

Remarks

The version without an execution policy is effectively the same as:

```
C++
```

```
for (; first != last; ++first)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type();
```

If an exception is thrown, previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified *`policy`*.

If a memory allocation failure occurs, a `std::bad_alloc` exception is thrown.

These functions are new in C++17.

`uninitialized_value_construct_n`

Constructs a specified number of objects of the iterator's `value_type` by value initialization, starting at the specified location.

C++

```
template <class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(
    ForwardIterator first,
    Size count);

template <class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(
    ExecutionPolicy&& policy,
    ForwardIterator first,
    Size count);
```

Parameters

`policy`

The execution policy to use.

`first`

An iterator addressing the first element in the destination range to construct.

`count`

The count of elements in the destination range to construct.

Remarks

The version without an execution policy is effectively the same as:

C++

```
for (; count > 0; (void)++first, --count)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type();
return first;
```

If an exception is thrown, previously constructed objects are destroyed in unspecified order.

The version with an execution policy has the same result, but executes according to the specified `policy`.

If a memory allocation failure occurs, a `std::bad_alloc` exception is thrown.

These functions are new in C++17.

`uses_allocator_v`

A helper variable template to access the value of the `uses_allocator` template.

C++

```
template <class T, class Alloc>
inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;
```

See also

<memory>

`<memory>` operators

Article • 12/07/2021

operator!=

Tests for inequality between objects.

```
C++
```

```
template <class Type, class Other>
bool operator!=(
    const allocator<Type>& left,
    const allocator<Other>& right) throw();

template <class T, class Del1, class U, class Del2>
bool operator!=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U&, Del2>& right);

template <class Ty1, class Ty2>
bool operator!=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left
One of the objects to be tested for inequality.

right
One of the objects to be tested for inequality.

Ty1
The type controlled by the left shared pointer.

Ty2
The type controlled by the right shared pointer.

Return Value

`true` if the objects are not equal; `false` if objects are equal.

Remarks

The first template operator returns false. (All default allocators are equal.)

The second and third template operators return `!(left == right)`.

Example

The first code example shows the results of comparing allocator objects.

C++

```
// memory_op_me.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<double> Alloc;
   vector <char>:: allocator_type v1Alloc;

   if ( Alloc != v1Alloc )
      cout << "The allocator objects Alloc & v1Alloc not are equal."
           << endl;
   else
      cout << "The allocator objects Alloc & v1Alloc are equal."
           << endl;
}
```

Output

```
The allocator objects Alloc & v1Alloc are equal.
```

The next code example compares shared pointers to allocated objects:

C++

```
// std__memory__operator_ne.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
    {
    std::shared_ptr<int> sp0(new int(0));
    std::shared_ptr<int> sp1(new int(0));
```

```
    std::cout << "sp0 != sp0 == " << std::boolalpha
        << (sp0 != sp0) << std::endl;
    std::cout << "sp0 != sp1 == " << std::boolalpha
        << (sp0 != sp1) << std::endl;

    return (0);
    }
```

Output

```
sp0 != sp0 == false
sp0 != sp1 == true
```

operator==

Tests for equality between objects.

C++

```
template <class Type, class Other>
bool operator==(
    const allocator<Type>& left,
    const allocator<Other>& right) throw();

template <class Ty1, class Del1, class Ty2, class Del2>
bool operator==(
    const unique_ptr<Ty1, Del1>& left,
    const unique_ptr<Ty2, Del2>& right);

template <class Ty1, class Ty2>
bool operator==(
    const shared_ptr<Ty1>& left;,
    const shared_ptr<Ty2>& right);
```

Parameters

left
One of the objects to be tested for equality.

right
One of the objects to be tested for equality.

Ty1
The type controlled by the left shared pointer.

Ty2
The type controlled by the right shared pointer.

Return Value

`true` if the objects are equal, **`false`** if objects are not equal.

Remarks

The first template operator returns true. (All default allocators are equal.)

The second and third template operators return `left.get() == right.get()`.

Example

The first code example shows the results of comparing allocator objects constructed in different ways.

C++

```
// memory_op_eq.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<char> Alloc;
   vector <int>:: allocator_type v1Alloc;

   allocator<char> cAlloc(Alloc);
   allocator<int> cv1Alloc(v1Alloc);

   if ( cv1Alloc == v1Alloc )
      cout << "The allocator objects cv1Alloc & v1Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
           << endl;

   if ( cAlloc == Alloc )
      cout << "The allocator objects cAlloc & Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cAlloc & Alloc are not equal."
```

```
            << endl;
}
```

Output

```
The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.
```

The next code example compares shared pointers to allocated objects:

C++

```
// std__memory__operator_eq.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
    {
    std::shared_ptr<int> sp0(new int(0));
    std::shared_ptr<int> sp1(new int(0));

    std::cout << "sp0 == sp0 == " << std::boolalpha
        << (sp0 == sp0) << std::endl;
    std::cout << "sp0 == sp1 == " << std::boolalpha
        << (sp0 == sp1) << std::endl;

    return (0);
    }
```

Output

```
sp0 == sp0 == true
sp0 == sp1 == false
```

operator>=

Tests for one object being greater than or equal to a second object.

C++

```
template <class T, class Del1, class U, class Del2>
bool operator>=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U, Del2>& right);

template <class Ty1, class Ty2>
```

```
bool operator>=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left
One of the objects to be compared.

right
One of the objects to be compared.

Ty1
The type controlled by the left shared pointer.

Ty2
The type controlled by the right shared pointer.

Remarks

The template operators return `left.get() >= right.get()`.

`operator<`

Tests for one object being less than a second object.

C++

```
template <class T, class Del1, class U, class Del2>
bool operator<(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U&, Del2>& right);

template <class Ty1, class Ty2>
bool operator<(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left
One of the objects to be compared.

right
One of the objects to be compared.

Ty1
The type controlled by the left pointer.

Ty2
The type controlled by the right pointer.

`operator<=`

Tests for one object being less than or equal to a second object.

```
C++
```

```
template <class T, class Del1, class U, class Del2>
bool operator<=(
    const unique_ptr<T, Del1>& left,
    const unique_ptr<U&, Del2>& right);

template <class Ty1, class Ty2>
bool operator<=(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left
One of the objects to be compared.

right
One of the objects to be compared.

Ty1
The type controlled by the left shared pointer.

Ty2
The type controlled by the right shared pointer.

Remarks

The template operators return `left.get() <= right.get()`

operator>

Tests for one object being greater than a second object.

```
C++
```

```
template <class Ty1, class Del1, class Ty2, class Del2>
bool operator>(
    const unique_ptr<Ty1, Del1>& left,
    const unique_ptr<Ty2&, Del2gt;& right);

template <class Ty1, class Ty2>
bool operator>(
    const shared_ptr<Ty1>& left,
    const shared_ptr<Ty2>& right);
```

Parameters

left

One of the objects to be compared.

right

One of the objects to be compared.

Ty1

The type controlled by the left shared pointer.

Ty2

The type controlled by the right shared pointer.

operator<<

Writes the shared pointer to the stream.

```
C++
```

```
template <class Elem, class Tr, class Ty>
std::basic_ostream<Elem, Tr>& operator<<(std::basic_ostream<Elem, Tr>& out,
    shared_ptr<Ty>& sp);
```

Parameters

Elem

The type of the stream element.

Tr
The type the stream element traits.

Ty
The type controlled by the shared pointer.

out
The output stream.

sp
The shared pointer.

Remarks

The template function returns `out << sp.get()`.

Example

C++

```
// std__memory__operator_sl.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
    {
    std::shared_ptr<int> sp0(new int(5));

    std::cout << "sp0 == " << sp0 << " (varies)" << std::endl;

    return (0);
    }
```

Output

```
sp0 == 3f3040 (varies)
```

`<memory>` enums

Article • 12/07/2021

pointer_safety Enumeration

The enumeration of possible values returned by `get_pointer_safety`.

```
C++
```

```
class pointer_safety {
   relaxed,
   preferred,
   strict
};
```

Remarks

The scoped `enum` defines the values that can be returned by `get_pointer_safety()`:

`relaxed` -- pointers not safely derived (obviously pointers to declared or allocated objects) are treated the same as those safely derived.

`preferred` -- as before, but pointers not safely derived should not be dereferenced.

`strict` -- pointers not safely derived might be treated differently than those safely derived.

allocator Class

Article • 01/05/2022

The class template describes an object that manages storage allocation and freeing for arrays of objects of type `Type`. An object of class `allocator` is the default allocator object specified in the constructors for several container class templates in the C++ Standard Library.

Syntax

```
C++
```

```
template <class Type>
class allocator
```

Parameters

Type
The type of object for which storage is being allocated or deallocated.

Remarks

All the C++ Standard Library containers have a template parameter that defaults to `allocator`. Constructing a container with a custom allocator provides control over allocation and freeing of that container's elements.

For example, an allocator object might allocate storage on a private heap or in shared memory, or it might optimize for small or large object sizes. It might also specify, through the type definitions it supplies, that elements be accessed through special accessor objects that manage shared memory, or perform automatic garbage collection. Hence, a class that allocates storage using an allocator object should use these types for declaring pointer and reference objects, as the containers in the C++ Standard Library do.

(C++98/03 only) When you derive from allocator class, you have to provide a rebind struct, whose `_Other` typedef references your newly-derived class.

Thus, an allocator defines the following types:

- pointer behaves like a pointer to `Type`.

- const_pointer behaves like a const pointer to `Type`.
- reference behaves like a reference to `Type`.
- const_reference behaves like a const reference to `Type`.

These `Type`s specify the form that pointers and references must take for allocated elements. (allocator::pointer is not necessarily the same as `Type*` for all allocator objects, even though it has this obvious definition for class `allocator`.)

C++11 and later: To enable move operations in your allocator, use the minimal allocator interface and implement copy constructor, == and != operators, allocate and deallocate. For more information and an example, see Allocators

Members

Constructors

Name	Description
allocator	Constructors used to create `allocator` objects.

Typedefs

Name	Description
const_pointer	A type that provides a constant pointer to the type of object managed by the allocator.
const_reference	A type that provides a constant reference to type of object managed by the allocator.
difference_type	A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.
pointer	A type that provides a pointer to the type of object managed by the allocator.
reference	A type that provides a reference to the type of object managed by the allocator.
size_type	An unsigned integral type that can represent the length of any sequence that an object of type `allocator` can allocate.
value_type	A type that is managed by the allocator.

Functions

Name	Description
address	Finds the address of an object whose value is specified.
allocate	Allocates a block of memory large enough to store at least some specified number of elements.
construct	Constructs a specific type of object at a specified address that is initialized with a specified value.
deallocate	Frees a specified number of objects from storage beginning at a specified position.
destroy	Calls an objects destructor without deallocating the memory where the object was stored.
max_size	Returns the number of elements of type `Type` that could be allocated by an object of class `allocator` before the free memory is used up.
rebind	A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

Operators

Name	Description
operator=	Assigns one `allocator` object to another `allocator` object.

address

Finds the address of an object whose value is specified.

```
C++
```

```
pointer address(reference val) const;
const_pointer address(const_reference val) const;
```

Parameters

val
The const or nonconst value of the object whose address is being searched for.

Return Value

A const or nonconst pointer to the object found of, respectively, const or nonconst value.

Remarks

The member functions return the address of *val*, in the form that pointers must take for allocated elements.

Example

C++

```
// allocator_address.cpp
// compile with: /EHsc
#include <memory>
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 2 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 8;
   v1Ptr = v1Alloc.address( *find(v1.begin( ), v1.end( ), k) );
   // v1Ptr = v1Alloc.address( k );
   cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 8.
```

allocate

Allocates a block of memory large enough to store at least some specified number of elements.

C++

```
pointer allocate(size_type count, const void* _Hint);
```

Parameters

count
The number of elements for which sufficient storage is to be allocated.

_Hint
A const pointer that may assist the allocator object satisfy the request for storage by locating the address of an object allocated prior to the request.

Return Value

A pointer to the allocated object or null if memory was not allocated.

Remarks

The member function allocates storage for an array of count elements of type `Type`, by calling operator new(*count*). It returns a pointer to the allocated object. The hint argument helps some allocators in improving locality of reference; a valid choice is the address of an object earlier allocated by the same allocator object and not yet deallocated. To supply no hint, use a null pointer argument instead.

Example

C++

```
// allocator_allocate.cpp
// compile with: /EHsc
#include <memory>
```

```
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   allocator<int> v1Alloc;
   allocator<int>::pointer v1aPtr;
   v1aPtr = v1Alloc.allocate ( 10 );

   int i;
   for ( i = 0 ; i < 10 ; i++ )
   {
      v1aPtr[ i ] = i;
   }

   for ( i = 0 ; i < 10 ; i++ )
   {
      cout << v1aPtr[ i ] << " ";
   }
   cout << endl;
   v1Alloc.deallocate( v1aPtr, 10 );
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

allocator

Constructors used to create allocator objects.

C++

```
allocator();
allocator(const allocator<Type>& right);
template <class Other>
    allocator(const allocator<Other>& right);
```

Parameters

right
The allocator object to be copied.

Remarks

The constructor does nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example

C++

```
// allocator_allocator.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
   Int( int i )
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

int main( )
{
   allocator<double> Alloc;
   vector <Int>:: allocator_type v1Alloc;

   allocator<double> cAlloc(Alloc);
   allocator<Int> cv1Alloc(v1Alloc);

   if ( cv1Alloc == v1Alloc )
      cout << "The allocator objects cv1Alloc & v1Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cv1Alloc & v1Alloc are not equal."
```

```
            << endl;

   if ( cAlloc == Alloc )
      cout << "The allocator objects cAlloc & Alloc are equal."
           << endl;
   else
      cout << "The allocator objects cAlloc & Alloc are not equal."
           << endl;
}
```

Output

```
The allocator objects cv1Alloc & v1Alloc are equal.
The allocator objects cAlloc & Alloc are equal.
```

const_pointer

A type that provides a constant pointer to the type of object managed by the allocator.

C++

```
typedef const value_type *const_pointer;
```

Remarks

The pointer type describes an object `ptr` that can designate, through the expression `*ptr`, any const object that an object of type `allocator` can allocate.

Example

C++

```
// allocator_const_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;
```

```
   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( i * 2 );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 10;
   v1Ptr = v1Alloc.address( k );

   cout << "The integer's address found has a value of: "
        << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The integer's address found has a value of: 10.
```

const_reference

A type that provides a constant reference to type of object managed by the allocator.

C++

```
typedef const value_type& const_reference;
```

Remarks

The reference type describes an object that can designate any const object that an object of type `allocator` can allocate.

Example

C++

```
// allocator_const_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
```

```
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vfIter = v.begin( );
   allocator<double>::const_reference vcref =*vfIter;
   cout << "The value of the element referred to by vref is: "
        << vcref << ",\n the first element in the vector." << endl;

   // const references can have their elements modified,
   // so the following would generate an error:
   // vcref = 150;
   // but the value of the first element could be modified through
   // its nonconst iterator and the const reference would remain valid
*vfIter = 175;
   cout << "The value of the element referred to by vcref,"
        <<"\n after nofication through its nonconst iterator, is: "
        << vcref << "." << endl;
}
```

Output

```
The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
the first element in the vector.
The value of the element referred to by vcref,
after nofication through its nonconst iterator, is: 175.
```

construct

Constructs a specific type of object at a specified address that is initialized with a specified value.

C++

```
void construct(pointer ptr, const Type& val);
void construct(pointer ptr, Type&& val);
template <class _Other>
    void construct(pointer ptr, _Other&&... val);
```

Parameters

ptr

A pointer to the location where the object is to be constructed.

val

The value with which the object being constructed is to be initialized.

Remarks

The first member function is equivalent to `new ((void *) ptr) Type(val)`.

Example

C++

```
// allocator_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 3 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;
```

```
   allocator<int>::pointer v1PtrA;
   int kA = 6, kB = 7;
   v1PtrA = v1Alloc.address( *find( v1.begin( ), v1.end( ), kA ) );
   v1Alloc.destroy ( v1PtrA );
   v1Alloc.construct ( v1PtrA , kB );

   cout << "The modified vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
( 3 6 9 12 15 18 21 ).
The modified vector v1 is:
( 3 7 9 12 15 18 21 ).
```

deallocate

Frees a specified number of objects from storage beginning at a specified position.

C++

```
void deallocate(pointer ptr, size_type count);
```

Parameters

ptr
A pointer to the first object to be deallocated from storage.

count
The number of objects to be deallocated from storage.

Remarks

The member function frees storage for the array of count objects of type `Type` beginning at *ptr*, by calling `operator delete(ptr)`. The pointer *ptr* must have been returned earlier by a call to allocate for an allocator object that compares equal to ***this**, allocating an array object of the same size and type. `deallocate` never throws an exception.

Example

For an example using the member function, see allocator::allocate.

destroy

Calls an objects destructor without deallocating the memory where the object was stored.

```
C++
```

```
void destroy(pointer ptr);
```

Parameters

ptr
A pointer designating the address of the object to be destroyed.

Remarks

The member function destroys the object designated by *ptr*, by calling the destructor `ptr->Type::~Type`.

Example

```
C++
```

```
// allocator_destroy.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
```

```
      v1.push_back( 2 * i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::pointer v1PtrA;
   int kA = 12, kB = -99;
   v1PtrA = v1Alloc.address( *find(v1.begin( ), v1.end( ), kA) );
   v1Alloc.destroy ( v1PtrA );
   v1Alloc.construct ( v1PtrA , kB );

   cout << "The modified vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
( 2 4 6 8 10 12 14 ).
The modified vector v1 is:
( 2 4 6 8 10 -99 14 ).
```

difference_type

A signed integral type that can represent the difference between values of pointers to the type of object managed by the allocator.

C++

```
typedef ptrdiff_t difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of type `allocator` can allocate.

Example

C++

```
// allocator_diff_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 0 ; i <= 7 ; i++ )
   {
      v1.push_back( i * 2 );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::const_pointer v1PtrA, v1PtrB;
   const int kA = 4, kB =12;
   v1PtrA = v1Alloc.address( kA );
   v1PtrB = v1Alloc.address( kB );
   allocator<int>::difference_type v1diff = *v1PtrB - *v1PtrA;

   cout << "Pointer v1PtrA addresses " << *v1PtrA << "." << endl;
   cout << "Pointer v1PtrB addresses " << *v1PtrB <<  "." << endl;
   cout << "The difference between the integer's addresses is: "
        << v1diff << "." << endl;
}
```

Output

```
The original vector v1 is:
( 0 2 4 6 8 10 12 14 ).
Pointer v1PtrA addresses 4.
Pointer v1PtrB addresses 12.
The difference between the integer's addresses is: 8.
```

max_size

Returns the number of elements of type `Type` that could be allocated by an object of class allocator before the free memory is used up.

C++

```
size_type max_size() const;
```

Return Value

The number of elements that could be allocated.

Example

C++

```
// allocator_max_size.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( i );
   }

   cout << "The original vector v1 is:\n ( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   vector <double> v2;
   vector <double> ::iterator v2Iter;
   vector <double> :: allocator_type v2Alloc;
   allocator<int>::size_type v1size;
   v1size = v1Alloc.max_size( );

   cout << "The number of integers that can be allocated before\n"
        << " the free memory in the vector v1 is used up is: "
        << v1size << "." << endl;

   int ii;
   for ( ii = 1 ; ii <= 7 ; ii++ )
   {
```

```
      v2.push_back( ii * 10.0 );
   }

   cout << "The original vector v2 is:\n ( " ;
   for ( v2Iter = v2.begin( ) ; v2Iter != v2.end( ) ; v2Iter++ )
      cout << *v2Iter << " ";
   cout << ")." << endl;
   allocator<double>::size_type v2size;
   v2size = v2Alloc.max_size( );

   cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v2 is used up is: "
        << v2size << "." << endl;
}
```

operator=

Assigns one allocator object to another allocator object.

C++

```
template <class Other>
    allocator<Type>& operator=(const allocator<Other>& right);
```

Parameters

right
An allocator object to be assigned to another such object.

Return Value

A reference to the allocator object

Remarks

The template assignment operator does nothing. In general, however, an allocator object assigned to another allocator object should compare equal to it and permit intermixing of object allocation and freeing between the two allocator objects.

Example

C++

```
// allocator_op_assign.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int {
public:
   Int(int i)
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( ) {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

int main( )
{
   allocator<Int> Alloc;
   allocator<Int> cAlloc ;
   cAlloc = Alloc;
}
```

pointer

A type that provides a pointer to the type of object managed by the allocator.

C++

```
typedef value_type *pointer;
```

Remarks

The pointer type describes an object `ptr` that can designate, through the expression ***ptr**, any object that an object of type `allocator` can allocate.

Example

C++

```
// allocator_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <int> v1;
   vector <int>::iterator v1Iter;
   vector <int>:: allocator_type v1Alloc;

   int i;
   for ( i = 1 ; i <= 7 ; i++ )
   {
      v1.push_back( 3 * i );
   }

   cout << "The original vector v1 is:\n( " ;
   for ( v1Iter = v1.begin( ) ; v1Iter != v1.end( ) ; v1Iter++ )
      cout << *v1Iter << " ";
   cout << ")." << endl;

   allocator<int>::const_pointer v1Ptr;
   const int k = 12;
   v1Ptr = v1Alloc.address( k );

   cout << "The integer addressed by v1Ptr has a value of: "
        << "*v1Ptr = " << *v1Ptr << "." << endl;
}
```

Output

```
The original vector v1 is:
( 3 6 9 12 15 18 21 ).
The integer addressed by v1Ptr has a value of: *v1Ptr = 12.
```

rebind

A structure that enables an allocator for objects of one type to allocate storage for objects of another type.

```
C++
```

```
struct rebind { typedef allocator<_Other> other; };
```

Parameters

other
The type of element for which memory is being allocated.

Remarks

This structure is useful for allocating memory for type that differs from the element type of the container being implemented.

The member class template defines the type other. Its sole purpose is to provide the type name `allocator<_Other>`, given the type name `allocator<Type>`.

For example, given an allocator object `al` of type `A`, you can allocate an object of type `_Other` with the expression:

```
C++
```

```
A::rebind<Other>::other(al).allocate(1, (Other *)0)
```

Or, you can name its pointer type by writing the type:

```
C++
```

```
A::rebind<Other>::other::pointer
```

Example

```
C++
```

```
// allocator_rebind.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <algorithm>
#include <vector>
```

```
using namespace std;

typedef vector<int>::allocator_type IntAlloc;
int main( )
{
   IntAlloc v1Iter;
   vector<int> v1;

   IntAlloc::rebind<char>::other::pointer pszC =
      IntAlloc::rebind<char>::other(v1.get_allocator()).allocate(1, (void
*)0);

   int * pInt = v1Iter.allocate(10);
}
```

reference

A type that provides a reference to the type of object managed by the allocator.

C++

```
typedef value_type& reference;
```

Remarks

The reference type describes an object that can designate any object that an object of type `allocator` can allocate.

Example

C++

```
// allocator_reference.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;
```

```
   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vfIter = v.begin( );
   allocator<double>::reference vref =*vfIter;
   cout << "The value of the element referred to by vref is: "
        << vref << ",\n the first element in the vector." << endl;

   // nonconst references can have their elements modified
   vref = 150;
   cout << "The element referred to by vref after being modified is: "
        << vref << "." << endl;
}
```

Output

```
The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element referred to by vref is: 100,
the first element in the vector.
The element referred to by vref after being modified is: 150.
```

size_type

An unsigned integral type that can represent the length of any sequence that an object of type `allocator` can allocate.

C++

```
typedef size_t size_type;
```

Example

C++

```
// allocator_size_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
```

```
#include <vector>

using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   allocator<double>::size_type vsize;
   vsize = vAlloc.max_size( );

   cout << "The number of doubles that can be allocated before\n"
        << " the free memory in the vector v is used up is: "
        << vsize << "." << endl;
}
```

value_type

A type that is managed by the allocator.

C++

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.

Example

C++

```
// allocator_value_type.cpp
// compile with: /EHsc
#include <memory>
```

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
   vector <double> v;
   vector <double> ::iterator vIter, vfIter;
   vector <double> :: allocator_type vAlloc;

   int j;
   for ( j = 1 ; j <= 7 ; j++ )
   {
      v.push_back( 100.0 * j );
   }

   cout << "The original vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;

   vfIter = v.begin( );
   allocator<double>::value_type vecVal = 150.0;
*vfIter = vecVal;
   cout << "The value of the element addressed by vfIter is: "
        << *vfIter << ",\n the first element in the vector." << endl;

   cout << "The modified vector v is:\n ( " ;
   for ( vIter = v.begin( ) ; vIter != v.end( ) ; vIter++ )
      cout << *vIter << " ";
   cout << ")." << endl;
}
```

Output

```
The original vector v is:
( 100 200 300 400 500 600 700 ).
The value of the element addressed by vfIter is: 150,
the first element in the vector.
The modified vector v is:
( 150 200 300 400 500 600 700 ).
```

Helpers

allocator_arg_t

C++

```
struct allocator_arg_t { explicit allocator_arg_t() = default; };
inline constexpr allocator_arg_t allocator_arg{};
```

uses_allocator

C++

```
template <class T, class Alloc> struct uses_allocator;
```

`allocator<void>` Class

Article • 12/06/2021

A specialization of the class template allocator to type `void`, defining the types that make sense in this context.

Syntax

C++

```
template <>
class allocator<void> {
    typedef void *pointer;
    typedef const void *const_pointer;
    typedef void value_type;
    template <class Other>
    struct rebind;
    allocator();
    allocator(const allocator<void>&);

    template <class Other>
    allocator(const allocator<Other>&);

    template <class Other>
    allocator<void>& operator=(const allocator<Other>&);
};
```

Remarks

The class explicitly specializes class template allocator for type `void`. Its constructors and assignment operator behave the same as for the class template, but it defines only the following types:

- const_pointer.
- pointer.
- value_type.
- rebind, a nested class template.

allocator_traits Class

Article • 08/03/2021

The class template describes an object that supplements an *allocator type*. An allocator type is any type that describes an allocator object that is used for managing allocated storage. Specifically, for any allocator type `Alloc`, you can use `allocator_traits<Alloc>` to determine all the information that is needed by an allocator-enabled container. For more information, see the default allocator Class.

Syntax

```
C++
```

```
template <class Alloc>
    class allocator_traits;
```

Members

Typedefs

Name	Description
`allocator_type`	This type is a synonym for the template parameter `Alloc`.
`const_pointer`	This type is `Alloc::const_pointer`, if that type is well-formed; otherwise, this type is `pointer_traits<pointer>::rebind<const value_type>`.
`const_void_pointer`	This type is `Alloc::const_void_pointer`, if that type is well-formed; otherwise, this type is `pointer_traits<pointer>::rebind<const void>`.
`difference_type`	This type is `Alloc::difference_type`, if that type is well-formed; otherwise, this type is `pointer_traits<pointer>::difference_type`.
`pointer`	This type is `Alloc::pointer`, if that type is well-formed; otherwise, this type is `value_type *`.

Name	Description
`propagate_on_container_copy_assignment`	This type is `Alloc::propagate_on_container_copy_assignment`, if that type is well-formed; otherwise, this type is `false_type`.
`propagate_on_container_move_assignment`	This type is `Alloc::propagate_on_container_move_assignment`, if that type is well-formed; otherwise, this type is `false_type`. If the type holds true, an allocator-enabled container copies its stored allocator on a move assignment.
`propagate_on_container_swap`	This type is `Alloc::propagate_on_container_swap`, if that type is well-formed; otherwise, this type is `false_type`. If the type holds true, an allocator-enabled container swaps its stored allocator on a swap.
`size_type`	This type is `Alloc::size_type`, if that type is well-formed; otherwise, this type is `make_unsigned<difference_type>::type`.
`value_type`	This type is a synonym for `Alloc::value_type`.
`void_pointer`	This type is `Alloc::void_pointer`, if that type is well-formed; otherwise, this type is `pointer_traits<pointer>::rebind<void>`.

Static Methods

The following static methods call the corresponding method on a given allocator parameter.

Name	Description
allocate	Static method that allocates memory by using the given allocator parameter.
construct	Static method that uses a specified allocator to construct an object.
deallocate	Static method that uses a specified allocator to deallocate a specified number of objects.
destroy	Static method that uses a specified allocator to call the destructor on an object without deallocating its memory.

Name	Description
max_size	Static method that uses a specified allocator to determine the maximum number of objects that can be allocated.
select_on_container_copy_construction	Static method that calls `select_on_container_copy_construction` on the specified allocator.

allocate

Static method that allocates memory by using the given allocator parameter.

```
C++
```

```
static pointer allocate(Alloc& al, size_type count);

static pointer allocate(Alloc& al, size_type count,
    typename allocator_traits<void>::const_pointer* hint);
```

Parameters

al
An allocator object.

count
The number of elements to allocate.

hint
A `const_pointer` that might assist the allocator object in satisfying the request for storage by locating the address of an allocated object prior to the request. A null pointer is treated as no hint.

Return Value

Each method returns a pointer to the allocated object.

The first static method returns `al.allocate(count)`.

The second method returns `al.allocate(count, hint)`, if that expression is well formed; otherwise it returns `al.allocate(count)`.

construct

Static method that uses a specified allocator to construct an object.

```
C++
```

```
template <class Uty, class Types>
static void construct(Alloc& al, Uty* ptr, Types&&... args);
```

Parameters

al
An allocator object.

ptr
A pointer to the location where the object is to be constructed.

args
A list of arguments that is passed to the object constructor.

Remarks

The static member function calls `al.construct(ptr, args...)`, if that expression is well formed; otherwise it evaluates `::new (static_cast<void *>(ptr)) Uty(std::forward<Types>(args)...)`.

deallocate

Static method that uses a specified allocator to deallocate a specified number of objects.

```
C++
```

```
static void deallocate(Alloc al,
    pointer ptr,
    size_type count);
```

Parameters

al
An allocator object.

ptr
A pointer to the starting location of the objects to be deallocated.

count
The number of objects to deallocate.

Remarks

This method calls `al.deallocate(ptr, count)`.

This method throws nothing.

destroy

Static method that uses a specified allocator to call the destructor on an object without deallocating its memory.

```
C++
```

```
template <class Uty>
    static void destroy(Alloc& al, Uty* ptr);
```

Parameters

al
An allocator object.

ptr
A pointer to the location of the object.

Remarks

This method calls `al.destroy(ptr)`, if that expression is well formed; otherwise it evaluates `ptr->~Uty()`.

max_size

Static method that uses a specified allocator to determine the maximum number of objects that can be allocated.

```
C++
```

```
static size_type max_size(const Alloc& al);
```

Parameters

al
An allocator object.

Remarks

This method returns `al.max_size()`, if that expression is well formed; otherwise it returns `numeric_limits<size_type>::max()`.

select_on_container_copy_construction

Static method that calls `select_on_container_copy_construction` on the specified allocator.

C++

```
static Alloc select_on_container_copy_construction(const Alloc& al);
```

Parameters

al
An allocator object.

Return Value

This method returns `al.select_on_container_copy_construction()`, if that type is well formed; otherwise it returns *al*.

Remarks

This method is used to specify an allocator when the associated container is copy-constructed.

auto_ptr class

Article • 11/02/2023

Wraps a smart pointer around a resource that ensures the resource is destroyed automatically when control leaves a block.

Starting in C++11, use `unique_ptr` instead of `auto_ptr`. For more information, see unique_ptr class. `auto_ptr` was deprecated in C++11 and removed in C++17.

For more information about `throw()` and exception handling, see Exception Specifications (throw).

Syntax

C++

```
class auto_ptr {
    typedef Type element_type;
    explicit auto_ptr(Type* ptr = 0) throw();
    auto_ptr(auto_ptr<Type>& right) throw()
        ;
    template <class Other>
    operator auto_ptr<Other>() throw();
    template <class Other>
    auto_ptr<Type>& operator=(auto_ptr<Other>& right) throw();
    template <class Other>
    auto_ptr(auto_ptr<Other>& right);
    auto_ptr<Type>& operator=(auto_ptr<Type>& right);
    ~auto_ptr();
    Type& operator*() const throw();
    Type * operator->()const throw();
    Type *get() const throw();
    Type *release()throw();
    void reset(Type* ptr = 0);
};
```

Parameters

right

The `auto_ptr` from which to get an existing resource.

ptr

The pointer specified to replace the stored pointer.

Remarks

The class template describes a smart pointer, called an `auto_ptr`, to an allocated object. The pointer must be either null or designate an object allocated by `new`. The `auto_ptr` transfers ownership if its stored value is assigned to another object. (It replaces the stored value after a transfer with a null pointer.) The destructor for `auto_ptr<Type>` deletes the allocated object. The `auto_ptr<Type>` ensures that an allocated object is automatically deleted when control leaves a block, even through a thrown exception. You shouldn't construct two `auto_ptr<Type>` objects that own the same object.

You can pass an `auto_ptr<Type>` object by value as an argument to a function call. An `auto_ptr` can't be an element of any Standard Library container. You can't reliably manage a sequence of `auto_ptr<Type>` objects with a C++ Standard Library container.

Members

Constructors

Name	Description
auto_ptr	The constructor for objects of type `auto_ptr`.

Typedefs

Name	Description
element_type	The type is a synonym for the template parameter `Type`.

Functions

Name	Description
get	The member function returns the stored pointer `myptr`.
release	The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.
reset	The member function evaluates the expression `delete myptr`, but only if the stored pointer value `myptr` changes as a result of function call. It then replaces the stored pointer with *`ptr`*.

Operators

Name	Description
operator=	An assignment operator that transfers ownership from one `auto_ptr` object to another.
operator*	The dereferencing operator for objects of type `auto_ptr`.
operator->	The operator for allowing member access.
operator auto_ptr<Other>	Casts from one kind of `auto_ptr` to another kind of `auto_ptr`.
operator auto_ptr_ref<Other>	Casts from an `auto_ptr` to an `auto_ptr_ref`.

auto_ptr

The constructor for objects of type `auto_ptr`.

```
C++
```

```
explicit auto_ptr(Type* ptr  = 0) throw();

auto_ptr(auto_ptr<Type>& right) throw();

auto_ptr(auto _ptr_ref<Type> right) throw();

template <class Other>
auto _ptr(auto _ptr<Other>& right) throw();
```

Parameters

ptr

The pointer to the object that `auto_ptr` encapsulates.

right

The `auto_ptr` object to be copied by the constructor.

Remarks

The first constructor stores *ptr* in `myptr`, the stored pointer to the allocated object. The second constructor transfers ownership of the pointer stored in *right*, by storing *right*. release in `myptr`.

The third constructor behaves the same as the second, except that it stores `right`. `ref`. `release` in `myptr`, where `ref` is the reference stored in `right`.

The template constructor behaves the same as the second constructor, if a pointer to `Other` can be implicitly converted to a pointer to `Type`.

Example

C++

```
// auto_ptr_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
   Int(int i)
   {
      cout << "Constructing " << ( void* )this  << endl;
      x = i;
      bIsConstructed = true;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << endl;
      bIsConstructed = false;
   };
   Int &operator++( )
   {
      x++;
      return *this;
   };
   int x;
private:
   bool bIsConstructed;
};

void function ( auto_ptr<Int> &pi )
{
   ++( *pi );
   auto_ptr<Int> pi2( pi );
   ++( *pi2 );
   pi = pi2;
}

int main( )
```

```
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   cout << pi->x << endl;
   function( pi );
   cout << pi->x << endl;
}
```

Output

```
Constructing 00311AF8
5
7
Destructing 00311AF8
```

element_type

The type is a synonym for the template parameter `Type`.

C++

```
typedef Type element  _type;
```

get

The member function returns the stored pointer `myptr`.

C++

```
Type *get() const throw();
```

Return Value

The stored pointer `myptr`.

Example

C++

```
// auto_ptr_get.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
```

```
using namespace std;

class Int
{
public:
   Int(int i)
   {
      x = i;
      cout << "Constructing " << ( void* )this  << " Value: " << x << endl;
   };
   ~Int( )
   {
      cout << "Destructing " << ( void* )this << " Value: " << x << endl;
   };

   int x;

};

int main( )
{
   auto_ptr<Int> pi ( new Int( 5 ) );
   pi.reset( new Int( 6 ) );
   Int* pi2 = pi.get ( );
   Int* pi3 = pi.release ( );
   if (pi2 == pi3)
      cout << "pi2 == pi3" << endl;
   delete pi3;
}
```

Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

operator=

An assignment operator that transfers ownership from one `auto_ptr` object to another.

C++

```
template <class Other>
    auto_ptr<Type>& operator=(auto_ptr<Other>& right) throw();
auto_ptr<Type>& operator=(auto_ptr<Type>& right) throw();
auto_ptr<Type>& operator=(auto_ptr_ref<Type> right) throw();
```

Parameters

right

An object of type auto_ptr.

Return Value

A reference to an object of type auto_ptr<Type>.

Remarks

The assignment evaluates the expression delete myptr, but only if the stored pointer myptr changes as a result of the assignment. It then transfers ownership of the pointer stored in *right*, by storing *right*.release in myptr. The function returns *this.

Example

For an example of the use of the member operator, see auto_ptr.

operator*

The dereferencing operator for objects of type auto_ptr.

```
C++
```

```
Type& operator*() const throw();
```

Return Value

A reference to an object of type Type that the pointer owns.

Remarks

The indirection operator returns *get. Hence, the stored pointer must not be null.

Example

For an example of how to use the member function, see auto_ptr.

`operator->`

The operator for allowing member access.

```
C++
```

```
Type * operator->() const throw();
```

Return Value

A member of the object that `auto_ptr` owns.

Remarks

The selection operator returns `get( )`, so that the expression `ap->member` behaves the same as (`ap`. `get`())-> `member`, where `ap` is an object of class `auto_ptr< Type >`. Hence, the stored pointer must not be null, and `Type` must be a class, struct, or union type with a `member` member.

Example

For an example of how to use the member function, see auto_ptr.

`operator auto_ptr<Other>`

Casts from one kind of `auto_ptr` to another kind of `auto_ptr`.

```
C++
```

```
template <class Other>
operator auto _ptr<Other>() throw();
```

Return Value

The type cast operator returns `auto_ptr<` **Other** `>(*this)`.

Example

```
C++
```

```
// auto_ptr_op_auto_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;
int main()
{
   auto_ptr<int> pi ( new int( 5 ) );
   auto_ptr<const int> pc = ( auto_ptr<const int> )pi;
}
```

`operator auto_ptr_ref<Other>`

Casts from an `auto_ptr` to an `auto_ptr_ref`.

C++

```
template <class Other>
operator auto _ptr  _ref<Other>() throw();
```

Return Value

The type cast operator returns **auto_ptr_ref**`<Other>(*this)`.

Example

C++

```
// auto_ptr_op_auto_ptr_ref.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class C {
public:
    C(int _i) : m_i(_i) {
    }
    ~C() {
        cout << "~C:  " << m_i << "\n";
    }
    C &operator =(const int &x) {
        m_i = x;
```

```
        return *this;
    }
    int m_i;
};
void f(auto_ptr<C> arg) {
};
int main()
{
    const auto_ptr<C> ciap(new C(1));
    auto_ptr<C> iap(new C(2));

    // Error: this implies transfer of ownership of iap's pointer
    // f(ciap);
    f(iap); // compiles, but gives up ownership of pointer

            // here, iap owns a destroyed pointer so the following is bad:
            // *iap = 5; // BOOM

    cout << "main exiting\n";
}
```

Output

```
~C:  2
main exiting
~C:  1
```

`release`

The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.

C++

```
Type *release() throw();
```

Return Value

The previously stored pointer.

Remarks

The member replaces the stored pointer `myptr` with a null pointer and returns the previously stored pointer.

Example

C++

```
// auto_ptr_release.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

class Int
{
public:
    Int(int i)
    {
        x = i;
        cout << "Constructing " << (void*)this << " Value: " << x << endl;
    };
    ~Int() {
        cout << "Destructing " << (void*)this << " Value: " << x << endl;
    };

    int x;

};

int main()
{
    auto_ptr<Int> pi(new Int(5));
    pi.reset(new Int(6));
    Int* pi2 = pi.get();
    Int* pi3 = pi.release();
    if (pi2 == pi3)
        cout << "pi2 == pi3" << endl;
    delete pi3;
}
```

Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

`reset`

The member function evaluates the expression `delete myptr`, but only if the stored pointer value `myptr` changes as a result of a function call. It then replaces the stored

pointer with `ptr`.

```
C++
```

```
void reset(Type* ptr = 0);
```

Parameters

`ptr`

The pointer specified to replace the stored pointer `myptr`.

Example

```
C++
```

```
// auto_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
#include <vector>

using namespace std;

class Int
{
public:
    Int(int i)
    {
        x = i;
        cout << "Constructing " << (void*)this << " Value: " << x << endl;
    };
    ~Int()
    {
        cout << "Destructing " << (void*)this << " Value: " << x << endl;
    };

    int x;
};

int main()
{
    auto_ptr<Int> pi(new Int(5));
    pi.reset(new Int(6));
    Int* pi2 = pi.get();
    Int* pi3 = pi.release();
    if (pi2 == pi3)
        cout << "pi2 == pi3" << endl;
```

```
    delete pi3;
}
```

Output

```
Constructing 00311AF8 Value: 5
Constructing 00311B88 Value: 6
Destructing 00311AF8 Value: 5
pi2 == pi3
Destructing 00311B88 Value: 6
```

See also

unique_ptr Class

bad_weak_ptr Class

Article • 08/03/2021

Reports bad weak_ptr exception.

Syntax

C++

```
class bad_weak_ptr : public std::exception
{
    bad_weak_ptr();
    const char *what() throw();
};
```

Remarks

The class describes an exception that can be thrown from the shared_ptr Class constructor that takes an argument of type weak_ptr Class. The member function `what` returns `"bad_weak_ptr"`.

Example

C++

```
// std__memory__bad_weak_ptr.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int);
        wp = sp;
    }

    try
    {
        std::shared_ptr<int> sp1(wp); // weak_ptr has expired
    }
    catch (const std::bad_weak_ptr&)
```

```
    {
        std::cout << "bad weak pointer" << std::endl;
    }
    catch (...)
    {
        std::cout << "unknown exception" << std::endl;
    }

    return (0);
}
```

Output

```
bad weak pointer
```

See also

weak_ptr Class

enable_shared_from_this Class

Article • 08/03/2021

Helps generate a `shared_ptr`.

Syntax

C++

```
class enable_shared_from_this {
public:
    shared_ptr<Ty>
        shared_from_this();
    shared_ptr<const Ty> shared_from_this() const;
    weak_ptr<T> weak_from_this() noexcept;
    weak_ptr<T const> weak_from_this() const noexcept;
protected:
    enable_shared_from_this();
    enable_shared_from_this(const enable_shared_from_this&);
    enable_shared_from_this& operator=(const enable_shared_from_this&);
    ~enable_shared_from_this();
};
```

Parameters

Ty
The type controlled by the shared pointer.

Remarks

Objects derived from `enable_shared_from_this` can use the `shared_from_this` methods in member functions to create shared_ptr owners of the instance that share ownership with existing `shared_ptr` owners. Otherwise, if you create a new `shared_ptr` by using **`this`**, it is distinct from existing `shared_ptr` owners, which can lead to invalid references or cause the object to be deleted more than once.

The constructors, destructor, and assignment operator are protected to help prevent accidental misuse. The template argument type *Ty* must be the type of the derived class.

For an example of usage, see enable_shared_from_this::shared_from_this.

shared_from_this

Generates a `shared_ptr` that shares ownership of the instance with existing `shared_ptr` owners.

C++

```
shared_ptr<T> shared_from_this();
shared_ptr<const T> shared_from_this() const;
```

Remarks

When you derive objects from the `enable_shared_from_this` base class, the `shared_from_this` template member functions return a shared_ptr Class object that shares ownership of this instance with existing `shared_ptr` owners. Otherwise, if you create a new `shared_ptr` from `this`, it is distinct from existing `shared_ptr` owners, which can lead to invalid references or cause the object to be deleted more than once. The behavior is undefined if you call `shared_from_this` on an instance that is not already owned by a `shared_ptr` object.

Example

C++

```
// std_memory_shared_from_this.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

using namespace std;

struct base : public std::enable_shared_from_this<base>
{
    int val;
    shared_ptr<base> share_more()
    {
        return shared_from_this();
    }
};

int main()
{
    auto sp1 = make_shared<base>();
    auto sp2 = sp1->share_more();

    sp1->val = 3;
```

```
    cout << "sp2->val == " << sp2->val << endl;
    return 0;
}
```

Output

```
sp2->val == 3
```

weak_from_this

C++

```
weak_ptr<T> weak_from_this() noexcept;
weak_ptr<T const> weak_from_this() const noexcept;
```

pointer_traits Struct

Article • 08/03/2021

Supplies information that is needed by an object of type `allocator_traits` to describe an allocator with pointer type `Ptr`.

Syntax

```
C++
```

```
template <class Ptr>
    struct pointer_traits;
```

Remarks

Ptr can be a raw pointer of type `Ty *` or a class with the following properties.

```
C++
```

```
struct Ptr
{ // describes a pointer type usable by allocators
   typedef Ptr pointer;
   typedef T1 element_type; // optional
   typedef T2 difference_type; // optional
   template <class Other>
   using rebind = typename Ptr<Other, Rest...>; // optional
   static pointer pointer_to(element_type& obj); // optional
};
```

Members

Typedefs

Name	Description
`typedef T2 difference_type`	The type `T2` is `Ptr::difference_type` if that type exists, otherwise `ptrdiff_t`. If `Ptr` is a raw pointer, the type is `ptrdiff_t`.
`typedef T1 element_type`	The type `T1` is `Ptr::element_type` if that type exists, otherwise `Ty`. If `Ptr` is a raw pointer, the type is `Ty`.

Name	Description
`typedef Ptr pointer`	The type is `Ptr`.

Structs

Name	Description
`rebind`	Attempts to convert the underlying pointer type to a specified type.

Methods

Name	Description
pointer_to	Converts an arbitrary reference to an object of class `Ptr`.

pointer_to

Static method that returns `Ptr::pointer_to(obj)`, if that function exists. Otherwise, it is not possible to convert an arbitrary reference to an object of class `Ptr`. If `Ptr` is a raw pointer, this method returns `addressof(obj)`.

C++

```
static pointer pointer_to(element_type& obj);
```

raw_storage_iterator Class

Article • 06/20/2022

An adaptor class that is provided to enable algorithms to store their results into uninitialized memory.

Syntax

```
C++
```

```
template <class OutputIterator, class Type>
    class raw_storage_iterator
```

Parameters

OutputIterator
Specifies the output iterator for the object being stored.

Type
The type of object for which storage is being allocated.

Remarks

The class describes an output iterator that constructs objects of type `Type` in the sequence it generates. An object of class `raw_storage_iterator`< **ForwardIterator**, **Type**> accesses storage through a forward iterator object, of class `ForwardIterator`, that you specify when you construct the object. For an object first of class `ForwardIterator`, the expression **&*first** must designate unconstructed storage for the next object (of type `Type`) in the generated sequence.

This adaptor class is used when it's necessary to separate memory allocation and object construction. The `raw_storage_iterator` can be used to copy objects into uninitialized storage, such as memory allocated using the `malloc` function.

Members

Constructors

Name	Description
raw_storage_iterator	Constructs a raw storage iterator with a specified underlying output iterator.

Typedefs

Name	Description
element_type	Provides a type that describes an element to be stored a raw storage iterator.
iter_type	Provides a type that describes an iterator that underlies a raw storage iterator.

Operators

Name	Description
operator*	A dereferencing operator used to implement the output iterator expression * `ii` = `x`.
operator=	An assignment operator used to implement the raw storage iterator expression * `i` = `x` for storing in memory.
operator++	Preincrement and postincrement operators for raw storage iterators.

element_type

Provides a type that describes an element to be stored a raw storage iterator.

```
C++
```

```
typedef Type element_type;
```

Remarks

The type is a synonym for the raw_storage_iterator class template parameter `Type`.

iter_type

Provides a type that describes an iterator that underlies a raw storage iterator.

```
C++
```

```
typedef ForwardIterator iter_type;
```

Remarks

The type is a synonym for the template parameter `ForwardIterator`.

operator*

A dereferencing operator used to implement the raw storage iterator expression * *ii* = *x*.

C++

```
raw_storage_iterator<ForwardIterator, Type>& operator*();
```

Return Value

A reference to the raw storage iterator

Remarks

The requirements for a `ForwardIterator` are that the raw storage iterator must satisfy, require only the expression * *ii* = *t* be valid and that it says nothing about the **`operator`** or the `operator=` on their own. The member operators in this implementation, returns **`*this`**, so that operator=(**constType**&) can perform the actual store in an expression, such as * *ptr* = `val`.

Example

C++

```
// raw_storage_iterator_op_deref.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int(int i)
   {
```

```
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };

   Int &operator=(int i)
   {
      if (!bIsConstructed)
         cout << "Not constructed.\n";
      cout << "Copying " << i << endl;
      x = i;
      return *this;
   };

   int x;

private:
   bool bIsConstructed;
};

int main( void)
{
   Int *pInt = ( Int* ) malloc( sizeof( Int ) );
   memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;
*pInt = 5;
   raw_storage_iterator< Int*, Int > it( pInt );
*it = 5;
}
```

Output

```
Not constructed.
Copying 5
Constructing 5
```

operator=

Assignment operator used to implement the raw storage iterator expression * *i* = *x* for storing in memory.

C++

```
raw_storage_iterator<ForwardIterator, Type>& operator=(
    const Type& val);
```

Parameters

val
The value of the object of type `Type` to be inserted into memory.

Return Value

The operator inserts `val` into memory, and then returns a reference to the raw storage iterator.

Remarks

The requirements for a `ForwardIterator` state that the raw storage iterator must satisfy, require only the expression * *ii* = *t* be valid, and that it says nothing about the **`operator`** or the `operator=` on their own. These member operators return **`*this`**.

The assignment operator constructs the next object in the output sequence using the stored iterator value `first`, by evaluating the placement new expression `new ( (void*) & *first ) Type( val )`.

Example

C++

```
// raw_storage_iterator_op_assign.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int( int i )
   {
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };
   Int &operator=( int i )
   {
      if ( !bIsConstructed )
         cout << "Not constructed.\n";
      cout << "Copying " << i << endl; x = i;
      return *this;
   };
   int x;
```

```
private:
   bool bIsConstructed;
};

int main( void )
{
   Int *pInt = ( Int* )malloc( sizeof( Int ) );
   memset( pInt, 0, sizeof( Int ) ); // Set bIsConstructed to false;

*pInt = 5;

   raw_storage_iterator<Int*, Int> it( pInt );
*it = 5;
}
```

Output

```
Not constructed.
Copying 5
Constructing 5
```

operator++

Preincrement and postincrement operators for raw storage iterators.

C++

```
raw_storage_iterator<ForwardIterator, Type>& operator++();

raw_storage_iterator<ForwardIterator, Type> operator++(int);
```

Return Value

A raw storage iterator or a reference to a raw storage iterator.

Remarks

The first operator eventually attempts to extract and store an object of type `CharType` from the associated input stream. The second operator makes a copy of the object, increments the object, and then returns the copy.

The first preincrement operator increments the stored output iterator object, and then returns `*this`.

The second postincrement operator makes a copy of `*this`, increments the stored output iterator object, and then returns the copy.

The constructor stores `first` as the output iterator object.

Example

C++

```
// raw_storage_iterator_op_incr.cpp
// compile with: /EHsc
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

int main( void )
{
   int *pInt = new int[5];
   std::raw_storage_iterator<int*,int> it( pInt );
   for ( int i = 0; i < 5; i++, it++ ) {
      *it = 2 * i;
   };

   for ( int i = 0; i < 5; i++ ) cout << "array " << i << " = " << pInt[i]
<< endl;;

   delete[] pInt;
}
```

Output

```
array 0 = 0
array 1 = 2
array 2 = 4
array 3 = 6
array 4 = 8
```

raw_storage_iterator

Constructs a raw storage iterator with a specified underlying output iterator.

C++

```
explicit raw_storage_iterator(ForwardIterator first);
```

Parameters

first

The forward iterator that is to underlie the `raw_storage_iterator` object being constructed.

Example

C++

```
// raw_storage_iterator_ctor.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <iterator>
#include <memory>
#include <list>
using namespace std;

class Int
{
public:
   Int(int i)
   {
      cout << "Constructing " << i << endl;
      x = i;
      bIsConstructed = true;
   };
   Int &operator=( int i )
   {
      if (!bIsConstructed)
         cout << "Error! I'm not constructed!\n";
      cout << "Copying " << i << endl;  x = i; return *this;
   };
   int x;
   bool bIsConstructed;
};

int main( void )
{
   std::list<int> l;
   l.push_back( 1 );
   l.push_back( 2 );
   l.push_back( 3 );
   l.push_back( 4 );

   Int *pInt = (Int*)malloc(sizeof(Int)*l.size( ));
   memset (pInt, 0, sizeof(Int)*l.size( ));
   // Hack: make sure bIsConstructed is false

   std::copy( l.begin( ), l.end( ), pInt );  // C4996
   for (unsigned int i = 0; i < l.size( ); i++)
      cout << "array " << i << " = " << pInt[i].x << endl;;
```

```
   memset (pInt, 0, sizeof(Int)*l.size( ));
   // hack: make sure bIsConstructed is false

   std::copy( l.begin( ), l.end( ),
      std::raw_storage_iterator<Int*,Int>(pInt));  // C4996
   for (unsigned int i = 0; i < l.size( ); i++ )
      cout << "array " << i << " = " << pInt[i].x << endl;

   free(pInt);
}
```

Output

```
Error! I'm not constructed!
Copying 1
Error! I'm not constructed!
Copying 2
Error! I'm not constructed!
Copying 3
Error! I'm not constructed!
Copying 4
array 0 = 1
array 1 = 2
array 2 = 3
array 3 = 4
Constructing 1
Constructing 2
Constructing 3
Constructing 4
array 0 = 1
array 1 = 2
array 2 = 3
array 3 = 4
```

`shared_ptr` class

Article • 06/21/2022

Wraps a reference-counted smart pointer around a dynamically allocated object.

Syntax

C++

```
template <class T>
class shared_ptr;
```

Remarks

The `shared_ptr` class describes an object that uses reference counting to manage resources. A `shared_ptr` object effectively holds a pointer to the resource that it owns or holds a null pointer. A resource can be owned by more than one `shared_ptr` object; when the last `shared_ptr` object that owns a particular resource is destroyed, the resource is freed.

A `shared_ptr` stops owning a resource when it's reassigned or reset.

The template argument `T` might be an incomplete type except as noted for certain member functions.

When a `shared_ptr<T>` object is constructed from a resource pointer of type `G*` or from a `shared_ptr<G>`, the pointer type `G*` must be convertible to `T*`. If it's not convertible, the code won't compile. For example:

C++

```
#include <memory>
using namespace std;

class F {};
class G : public F {};

shared_ptr<G> sp0(new G);   // okay, template parameter G and argument G*
shared_ptr<G> sp1(sp0);     // okay, template parameter G and argument
shared_ptr<G>
shared_ptr<F> sp2(new G);   // okay, G* convertible to F*
shared_ptr<F> sp3(sp0);     // okay, template parameter F and argument
shared_ptr<G>
```

```
shared_ptr<F> sp4(sp2);     // okay, template parameter F and argument
shared_ptr<F>
shared_ptr<int> sp5(new G); // error, G* not convertible to int*
shared_ptr<int> sp6(sp2);   // error, template parameter int and argument
shared_ptr<F>
```

A `shared_ptr` object owns a resource:

- if it was constructed with a pointer to that resource,
- if it was constructed from a `shared_ptr` object that owns that resource,
- if it was constructed from a weak_ptr object that points to that resource, or
- if ownership of that resource was assigned to it, either with shared_ptr::operator= or by calling the member function shared_ptr::reset.

The `shared_ptr` objects that own a resource share a control block. The control block holds:

- the number of `shared_ptr` objects that own the resource,
- the number of `weak_ptr` objects that point to the resource,
- the deleter for that resource if it has one,
- the custom allocator for the control block if it has one.

A `shared_ptr` object that is initialized by using a null pointer has a control block and isn't empty. After a `shared_ptr` object releases a resource, it no longer owns that resource. After a `weak_ptr` object releases a resource, it no longer points to that resource.

When the number of `shared_ptr` objects that own a resource becomes zero, the resource is freed, either by deleting it or by passing its address to a deleter, depending on how ownership of the resource was originally created. When the number of `shared_ptr` objects that own a resource is zero, and the number of `weak_ptr` objects that point to that resource is zero, the control block is freed, using the custom allocator for the control block if it has one.

An empty `shared_ptr` object doesn't own any resources and has no control block.

A deleter is a function object that has a member function `operator()`. Its type must be copy constructible, and its copy constructor and destructor must not throw exceptions. It accepts one parameter, the object to be deleted.

Some functions take an argument list that defines properties of the resulting `shared_ptr<T>` or `weak_ptr<T>` object. You can specify such an argument list in several ways:

no arguments: The resulting object is an empty `shared_ptr` object or an empty `weak_ptr` object.

`ptr`: A pointer of type `Other*` to the resource to be managed. `T` must be a complete type. If the function fails (because the control block can't be allocated), it evaluates the expression `delete ptr`.

`ptr, deleter`: A pointer of type `Other*` to the resource to be managed and a deleter for that resource. If the function fails (because the control block can't be allocated), it calls `deleter(ptr)`, which must be well-defined.

`ptr, deleter, alloc`: A pointer of type `Other*` to the resource to be managed, a deleter for that resource, and an allocator to manage any storage that must be allocated and freed. If the function fails (because the control block can't be allocated), it calls `deleter(ptr)`, which must be well-defined.

`sp`: A `shared_ptr<Other>` object that owns the resource to be managed.

`wp`: A `weak_ptr<Other>` object that points to the resource to be managed.

`ap`: An `auto_ptr<Other>` object that holds a pointer to the resource to be managed. If the function succeeds, it calls `ap.release()`; otherwise it leaves `ap` unchanged.

In all cases, the pointer type `Other*` must be convertible to `T*`.

Thread Safety

Multiple threads can read and write different `shared_ptr` objects at the same time, even when the objects are copies that share ownership.

Members

Name	Description
Constructors	
shared_ptr	Constructs a `shared_ptr`.
~shared_ptr	Destroys a `shared_ptr`.

Name	Description
Typedefs	
element_type	The type of an element.
weak_type	The type of a weak pointer to an element.
Member functions	
get	Gets address of owned resource.
owner_before	Returns true if this `shared_ptr` is ordered before (or less than) the provided pointer.
reset	Replace owned resource.
swap	Swaps two `shared_ptr` objects.
unique	Tests if owned resource is unique.
use_count	Counts numbers of resource owners.
Operators	
operator bool	Tests if an owned resource exists.
operator*	Gets the designated value.
operator=	Replaces the owned resource.
operator->	Gets a pointer to the designated value.

`element_type`

The type of an element.

```
C++
```

```
typedef T element_type;                  // before C++17
using element_type = remove_extent_t<T>; // C++17
```

Remarks

The `element_type` type is a synonym for the template parameter `T`.

Example

C++

```
// std__memory__shared_ptr_element_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::shared_ptr<int>::element_type val = *sp0;

    std::cout << "*sp0 == " << val << std::endl;

    return (0);
}
```

Output

```
*sp0 == 5
```

get

Gets address of owned resource.

C++

```
element_type* get() const noexcept;
```

Remarks

The member function returns the address of the owned resource. If the object doesn't own a resource, it returns 0.

Example

C++

```
// std__memory__shared_ptr_get.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>
```

```
int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));

    std::cout << "sp0.get() == 0 == " << std::boolalpha
        << (sp0.get() == 0) << std::endl;
    std::cout << "*sp1.get() == " << *sp1.get() << std::endl;

    return (0);
}
```

Output

```
sp0.get() == 0 == true
*sp1.get() == 5
```

`operator bool`

Tests if an owned resource exists.

C++

```
explicit operator bool() const noexcept;
```

Remarks

The operator returns a value of **`true`** when `get() != nullptr`, otherwise **`false`**.

Example

C++

```
// std__memory__shared_ptr_operator_bool.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));

    std::cout << "(bool)sp0 == " << std::boolalpha
        << (bool)sp0 << std::endl;
```

```
    std::cout << "(bool)sp1 == " << std::boolalpha
        << (bool)sp1 << std::endl;

    return (0);
}
```

Output

```
(bool)sp0 == false
(bool)sp1 == true
```

`operator*`

Gets the designated value.

C++

```
T& operator*() const noexcept;
```

Remarks

The indirection operator returns `*get()`. Hence, the stored pointer must not be null.

Example

C++

```
// std__memory__shared_ptr_operator_st.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));

    std::cout << "*sp0 == " << *sp0 << std::endl;

    return (0);
}
```

Output

```
*sp0 == 5
```

`operator=`

Replaces the owned resource.

```
C++
```

```
shared_ptr& operator=(const shared_ptr& sp) noexcept;

shared_ptr& operator=(shared_ptr&& sp) noexcept;

template <class Other>
shared_ptr& operator=(const shared_ptr<Other>& sp) noexcept;

template <class Other>
shared_ptr& operator=(shared_ptr<Other>&& sp) noexcept;

template <class Other>
shared_ptr& operator=(auto_ptr<Other>&& ap);    // deprecated in C++11,
removed in C++17

template <class Other, class Deleter>
shared_ptr& operator=(unique_ptr<Other, Deleter>&& up);
```

Parameters

sp

The shared pointer to copy or move from.

ap

The auto pointer to move. The `auto_ptr` overload is deprecated in C++11 and removed in C++17.

up

The unique pointer to the object to adopt ownership of. *up* owns no object after the call.

Other

The type of the object pointed to by *sp*, *ap*, or *up*.

Deleter

The type of the deleter of the owned object, stored for later deletion of the object.

Remarks

The operators all decrement the reference count for the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If the reference count falls to zero, the resource is released. If an operator fails, it leaves `*this` unchanged.

Example

C++

```
// std__memory__shared_ptr_operator_as.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0;
    std::shared_ptr<int> sp1(new int(5));
    std::unique_ptr<int> up(new int(10));

    sp0 = sp1;
    std::cout << "*sp0 == " << *sp0 << std::endl;

    sp0 = up;
    std::cout << "*sp0 == " << *sp0 << std::endl;

    return (0);
}
```

Output

```
*sp0 == 5
*sp0 == 10
```

`operator->`

Gets a pointer to the designated value.

C++

```
T* operator->() const noexcept;
```

Remarks

The selection operator returns `get()`, so that the expression `sp->member` behaves the same as `(sp.get())->member` where `sp` is an object of class `shared_ptr<T>`. Hence, the stored pointer must not be null, and `T` must be a class, structure, or union type with a member `member`.

Example

C++

```
// std__memory__shared_ptr_operator_ar.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

typedef std::pair<int, int> Mypair;
int main()
{
    std::shared_ptr<Mypair> sp0(new Mypair(1, 2));

    std::cout << "sp0->first == " << sp0->first << std::endl;
    std::cout << "sp0->second == " << sp0->second << std::endl;

    return (0);
}
```

Output

```
sp0->first == 1
sp0->second == 2
```

`owner_before`

Returns true if this `shared_ptr` is ordered before (or less than) the provided pointer.

C++

```
template <class Other>
bool owner_before(const shared_ptr<Other>& ptr) const noexcept;

template <class Other>
bool owner_before(const weak_ptr<Other>& ptr) const noexcept;
```

Parameters

ptr

An lvalue reference to either a `shared_ptr` or a `weak_ptr`.

Remarks

The template member function returns true if `*this` is ordered before `ptr`.

`reset`

Replace owned resource.

```
C++
```

```
void reset() noexcept;

template <class Other>
void reset(Other *ptr);

template <class Other, class Deleter>
void reset(
    Other *ptr,
    Deleter deleter);

template <class Other, class Deleter, class Allocator>
void reset(
    Other *ptr,
    Deleter deleter,
    Allocator alloc);
```

Parameters

Other

The type controlled by the argument pointer.

Deleter

The type of the deleter.

ptr

The pointer to copy.

deleter

The deleter to copy.

`Allocator`

The type of the allocator.

`alloc`

The allocator to copy.

Remarks

The operators all decrement the reference count for the resource currently owned by `*this` and assign ownership of the resource named by the operand sequence to `*this`. If the reference count falls to zero, the resource is released. If an operator fails, it leaves `*this` unchanged.

Example

C++

```
// std__memory__shared_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp(new int(5));

    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;

    sp.reset();
    std::cout << "(bool)sp == " << std::boolalpha
        << (bool)sp << std::endl;

    sp.reset(new int(10));
    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;

    sp.reset(new int(15), deleter());
    std::cout << "*sp == " << std::boolalpha
        << *sp << std::endl;
```

```
    return (0);
}
```

Output

```
*sp == 5
(bool)sp == false
*sp == 10
*sp == 15
```

shared_ptr

Constructs a `shared_ptr`.

C++

```
constexpr shared_ptr() noexcept;

constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() {}

shared_ptr(const shared_ptr& sp) noexcept;

shared_ptr(shared_ptr&& sp) noexcept;

template <class Other>
explicit shared_ptr(Other* ptr);

template <class Other, class Deleter>
shared_ptr(
    Other* ptr,
    Deleter deleter);

template <class Deleter>
shared_ptr(
    nullptr_t ptr,
    Deleter deleter);

template <class Other, class Deleter, class Allocator>
shared_ptr(
    Other* ptr,
    Deleter deleter,
    Allocator alloc);

template <class Deleter, class Allocator>
shared_ptr(
    nullptr_t ptr,
    Deleter deleter,
    Allocator alloc);

template <class Other>
```

```
shared_ptr(
    const shared_ptr<Other>& sp) noexcept;

template <class Other>
explicit shared_ptr(
    const weak_ptr<Other>& wp);

template <class &>
shared_ptr(
    std::auto_ptr<Other>& ap);

template <class &>
shared_ptr(
    std::auto_ptr<Other>&& ap);

template <class Other, class Deleter>
shared_ptr(
    unique_ptr<Other, Deleter>&& up);

template <class Other>
shared_ptr(
    const shared_ptr<Other>& sp,
    element_type* ptr) noexcept;

template <class Other>
shared_ptr(
    shared_ptr<Other>&& sp,
    element_type* ptr) noexcept;

template <class Other, class Deleter>
shared_ptr(
    const unique_ptr<Other, Deleter>& up) = delete;
```

Parameters

Other

The type controlled by the argument pointer.

ptr

The pointer to copy.

Deleter

The type of the deleter.

Allocator

The type of the allocator.

deleter

The deleter.

alloc

The allocator.

sp

The smart pointer to copy.

wp

The weak pointer.

ap

The auto pointer to copy.

Remarks

The constructors each construct an object that owns the resource named by the operand sequence. The constructor `shared_ptr(const weak_ptr<Other>& wp)` throws an exception object of type bad_weak_ptr if `wp.expired()`.

Example

C++

```
// std__memory__shared_ptr_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

struct deleter
{
    void operator()(int *p)
    {
        delete p;
    }
};

int main()
{
    std::shared_ptr<int> sp0;
    std::cout << "(bool)sp0 == " << std::boolalpha
        << (bool)sp0 << std::endl;

    std::shared_ptr<int> sp1(new int(5));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    std::shared_ptr<int> sp2(new int(10), deleter());
    std::cout << "*sp2 == " << *sp2 << std::endl;

    std::shared_ptr<int> sp3(sp2);
```

```
    std::cout << "*sp3 == " << *sp3 << std::endl;

    std::weak_ptr<int> wp(sp3);
    std::shared_ptr<int> sp4(wp);
    std::cout << "*sp4 == " << *sp4 << std::endl;

    std::auto_ptr<int> ap(new int(15));
    std::shared_ptr<int> sp5(ap);
    std::cout << "*sp5 == " << *sp5 << std::endl;

    return (0);
}
```

Output

```
(bool)sp0 == false
*sp1 == 5
*sp2 == 10
*sp3 == 10
*sp4 == 10
*sp5 == 15
```

~shared_ptr

Destroys a `shared_ptr`.

C++

```
~shared_ptr();
```

Remarks

The destructor decrements the reference count for the resource currently owned by `*this`. If the reference count falls to zero, the resource is released.

Example

C++

```
// std__memory__shared_ptr_destroy.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
```

```
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << "use count == " << sp1.use_count() << std::endl;

    {
        std::shared_ptr<int> sp2(sp1);
        std::cout << "*sp2 == " << *sp2 << std::endl;
        std::cout << "use count == " << sp1.use_count() << std::endl;
    }

    // check use count after sp2 is destroyed
    std::cout << "use count == " << sp1.use_count() << std::endl;

    return (0);
}
```

Output

```
*sp1 == 5
use count == 1
*sp2 == 5
use count == 2
use count == 1
```

swap

Swaps two `shared_ptr` objects.

C++

```
void swap(shared_ptr& sp) noexcept;
```

Parameters

sp

The shared pointer to swap with.

Remarks

The member function leaves the resource originally owned by `*this` subsequently owned by *sp*, and the resource originally owned by *sp* subsequently owned by `*this`. The function doesn't change the reference counts for the two resources and it doesn't throw any exceptions.

Example

C++

```
// std__memory__shared_ptr_swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}
```

Output

```
*sp1 == 5
*sp1 == 10
*sp1 == 5
*wp1 == 5
*wp1 == 10
*wp1 == 5
```

`unique`

Tests if owned resource is unique. This function was deprecated in C++17, and removed in C++20.

C++

```
bool unique() const noexcept;
```

Remarks

The member function returns `true` if no other `shared_ptr` object owns the resource that is owned by `*this`, otherwise `false`.

Example

C++

```
// std__memory__shared_ptr_unique.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "sp1.unique() == " << std::boolalpha
        << sp1.unique() << std::endl;

    std::shared_ptr<int> sp2(sp1);
    std::cout << "sp1.unique() == " << std::boolalpha
        << sp1.unique() << std::endl;

    return (0);
}
```

Output

```
sp1.unique() == true
sp1.unique() == false
```

use_count

Counts numbers of resource owners.

C++

```
long use_count() const noexcept;
```

Remarks

The member function returns the number of `shared_ptr` objects that own the resource that is owned by `*this`.

Example

C++

```
// std__memory__shared_ptr_use_count.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::cout << "sp1.use_count() == "
        << sp1.use_count() << std::endl;

    std::shared_ptr<int> sp2(sp1);
    std::cout << "sp1.use_count() == "
        << sp1.use_count() << std::endl;

    return (0);
}
```

Output

```
sp1.use_count() == 1
sp1.use_count() == 2
```

`weak_type`

The type of a weak pointer to an element.

C++

```
using weak_type = weak_ptr<T>; // C++17
```

Remarks

The `weak_type` definition was added in C++17.

See also

Header Files Reference

<memory>

unique_ptr

weak_ptr class

`unique_ptr` Class

Article • 06/21/2022

Stores a pointer to an owned object or array. The object/array is owned by no other `unique_ptr`. The object/array is destroyed when the `unique_ptr` is destroyed.

Syntax

C++

```
class unique_ptr {
public:
    unique_ptr();
    unique_ptr(nullptr_t Nptr);
    explicit unique_ptr(pointer Ptr);
    unique_ptr(pointer Ptr,
        typename conditional<is_reference<Del>::value, Del,
        typename add_reference<const Del>::type>::type Deleter);
    unique_ptr(pointer Ptr,
        typename remove_reference<Del>::type&& Deleter);
    unique_ptr(unique_ptr&& Right);
    template <class T2, Class Del2>
    unique_ptr(unique_ptr<T2, Del2>&& Right);
    unique_ptr(const unique_ptr& Right) = delete;
    unique_ptr& operator=(const unique_ptr& Right) = delete;
};

//Specialization for arrays:
template <class T, class D>
class unique_ptr<T[], D> {
public:
    typedef pointer;
    typedef T element_type;
    typedef D deleter_type;
    constexpr unique_ptr() noexcept;
    template <class U>
    explicit unique_ptr(U p) noexcept;
    template <class U>
    unique_ptr(U p, see below d) noexcept;
    template <class U>
    unique_ptr(U p, see below d) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept : unique_ptr() { }     template
<class U, class E>
        unique_ptr(unique_ptr<U, E>&& u) noexcept;
    ~unique_ptr();
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E>
    unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

```
    unique_ptr& operator=(nullptr_t) noexcept;
    T& operator[](size_t i) const;

    pointer get() const noexcept;
    deleter_type& get_deleter() noexcept;
    const deleter_type& get_deleter() const noexcept;
    explicit operator bool() const noexcept;
    pointer release() noexcept;
    void reset(pointer p = pointer()) noexcept;
    void reset(nullptr_t = nullptr) noexcept;
    template <class U>
    void reset(U p) noexcept = delete;
    void swap(unique_ptr& u) noexcept;  // disable copy from lvalue
unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
};
```

Parameters

Right

A `unique_ptr`.

Nptr

An `rvalue` of type `std::nullptr_t`.

Ptr

A `pointer`.

Deleter

A `deleter` function that is bound to a `unique_ptr`.

Exceptions

No exceptions are generated by `unique_ptr`.

Remarks

The `unique_ptr` class supersedes `auto_ptr`, and can be used as an element of C++ Standard Library containers.

Use the make_unique helper function to efficiently create new instances of `unique_ptr`.

`unique_ptr` uniquely manages a resource. Each `unique_ptr` object stores a pointer to the object that it owns or stores a null pointer. A resource can be owned by no more

than one `unique_ptr` object; when a `unique_ptr` object that owns a particular resource is destroyed, the resource is freed. A `unique_ptr` object may be moved, but not copied; for more information, see Rvalue Reference Declarator: &&.

The resource is freed by calling a stored `deleter` object of type `Del` that knows how resources are allocated for a particular `unique_ptr`. The default `deleter` `default_delete<T>` assumes that the resource pointed to by `ptr` is allocated with **`new`**, and that it can be freed by calling `delete _Ptr`. (A partial specialization `unique_ptr<T[]>`manages array objects allocated with `new[]`, and has the default `deleter` `default_delete<T[]>`, specialized to call `delete[] ptr`.)

The stored pointer to an owned resource, `stored_ptr` has type `pointer`. It's `Del::pointer` if defined, and `T *` if not. The stored `deleter` object `stored_deleter` occupies no space in the object if the `deleter` is stateless. Note that `Del` can be a reference type.

Members

Constructors

Name	Description
unique_ptr	There are seven constructors for `unique_ptr`.

Typedefs

Name	Description
deleter_type	A synonym for the template parameter `Del`.
element_type	A synonym for the template parameter `T`.
pointer	A synonym for `Del::pointer` if defined, otherwise `T *`.

Functions

Name	Description
get	Returns `stored_ptr`.
get_deleter	Returns a reference to `stored_deleter`.

Name	Description
release	stores `pointer()` in `stored_ptr` and returns its previous contents.
reset	Releases the currently owned resource and accepts a new resource.
swap	Exchanges resource and `deleter` with the provided `unique_ptr`.

Operators

Name	Description
`operator bool`	The operator returns a value of a type that is convertible to `bool`. The result of the conversion to `bool` is `true` when `get() != pointer()`, otherwise `false`.
`operator->`	The member function returns `stored_ptr`.
`operator*`	The member function returns `*stored_ptr`.
operator=	Assigns the value of a `unique_ptr` (or a `pointer-type`) to the current `unique_ptr`.

`deleter_type`

The type is a synonym for the template parameter `Del`.

```
C++

typedef Del deleter_type;
```

Remarks

The type is a synonym for the template parameter `Del`.

`element_type`

The type is a synonym for the template parameter `Type`.

```
C++

typedef Type element_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

`get`

Returns `stored_ptr`.

```
C++

pointer get() const;
```

Remarks

The member function returns `stored_ptr`.

`get_deleter`

Returns a reference to `stored_deleter`.

```
C++

Del& get_deleter();

const Del& get_deleter() const;
```

Remarks

The member function returns a reference to `stored_deleter`.

`operator=`

Assigns the address of the provided `unique_ptr` to the current one.

```
C++

unique_ptr& operator=(unique_ptr&& right);
template <class U, Class Del2>
unique_ptr& operator=(unique_ptr<Type, Del>&& right);
unique_ptr& operator=(pointer-type);
```

Parameters

A `unique_ptr` reference used to assign the value of to the current `unique_ptr`.

Remarks

The member functions call `reset(right.release())` and move `right.stored_deleter` to `stored_deleter`, then return `*this`.

`pointer`

A synonym for `Del::pointer` if defined, otherwise `Type *`.

```
C++
```

```
typedef T1 pointer;
```

Remarks

The type is a synonym for `Del::pointer` if defined, otherwise `Type *`.

`release`

Releases ownership of the returned stored pointer to the caller and sets the stored pointer value to `nullptr`.

```
C++
```

```
pointer release();
```

Remarks

Use `release` to take over ownership of the raw pointer stored by the `unique_ptr`. The caller is responsible for deletion of the returned pointer. The `unique-ptr` is set to the empty default-constructed state. You can assign another pointer of compatible type to the `unique_ptr` after the call to `release`.

Example

This example shows how the caller of release is responsible for the object returned:

C++

```cpp
// stl_release_unique.cpp
// Compile by using: cl /W4 /EHsc stl_release_unique.cpp
#include <iostream>
#include <memory>

struct Sample {
   int content_;
   Sample(int content) : content_(content) {
      std::cout << "Constructing Sample(" << content_ << ")" << std::endl;
   }
   ~Sample() {
      std::cout << "Deleting Sample(" << content_ << ")" << std::endl;
   }
};

void ReleaseUniquePointer() {
   // Use make_unique function when possible.
   auto up1 = std::make_unique<Sample>(3);
   auto up2 = std::make_unique<Sample>(42);

   // Take over ownership from the unique_ptr up2 by using release
   auto ptr = up2.release();
   if (up2) {
      // This statement does not execute, because up2 is empty.
      std::cout << "up2 is not empty." << std::endl;
   }
   // We are now responsible for deletion of ptr.
   delete ptr;
   // up1 deletes its stored pointer when it goes out of scope.
}

int main() {
   ReleaseUniquePointer();
}
```

Output

```
Constructing Sample(3)
Constructing Sample(42)
Deleting Sample(42)
Deleting Sample(3)
```

`reset`

Takes ownership of the pointer parameter, and then deletes the original stored pointer. If the new pointer is the same as the original stored pointer, `reset` deletes the pointer and sets the stored pointer to **`nullptr`**.

C++

```
void reset(pointer ptr = pointer());
void reset(nullptr_t ptr);
```

Parameters

ptr
A pointer to the resource to take ownership of.

Remarks

Use `reset` to change the stored pointer owned by the `unique_ptr` to *ptr* and then delete the original stored pointer. If the `unique_ptr` wasn't empty, `reset` invokes the deleter function returned by get_deleter on the original stored pointer.

Because `reset` first stores the new pointer *ptr*, and then deletes the original stored pointer, it's possible for `reset` to immediately delete *ptr* if it's the same as the original stored pointer.

`swap`

Exchanges pointers between two `unique_ptr` objects.

C++

```
void swap(unique_ptr& right);
```

Parameters

right
A `unique_ptr` used to swap pointers.

Remarks

The member function swaps `stored_ptr` with `right.stored_ptr` and `stored_deleter` with `right.stored_deleter`.

`unique_ptr`

There are seven constructors for `unique_ptr`.

```
C++
```

```
unique_ptr();

unique_ptr(nullptr_t);
explicit unique_ptr(pointer ptr);

unique_ptr(
    Type* ptr,
    typename conditional<
    is_reference<Del>::value,
    Del,
    typename add_reference<const Del>::type>::type _Deleter);

unique_ptr(pointer ptr, typename remove_reference<Del>::type&& _Deleter);
unique_ptr(unique_ptr&& right);
template <class Ty2, Class Del2>
    unique_ptr(unique_ptr<Ty2, Del2>&& right);
```

Parameters

`ptr`
A pointer to the resource to be assigned to a `unique_ptr`.

`_Deleter`
A `deleter` to be assigned to a `unique_ptr`.

`right`
An `rvalue reference` to a `unique_ptr` from which `unique_ptr` fields are move assigned to the newly constructed `unique_ptr`.

Remarks

The first two constructors construct an object that manages no resource. The third constructor stores *ptr* in `stored_ptr`. The fourth constructor stores *`ptr`* in `stored_ptr` and `deleter` in `stored_deleter`.

The fifth constructor stores *`ptr`* in `stored_ptr` and moves `deleter` into `stored_deleter`. The sixth and seventh constructors store `right.release()` in `stored_ptr` and moves `right.get_deleter()` into `stored_deleter`.

`~unique_ptr`

The destructor for `unique_ptr`, destroys a `unique_ptr` object.

```
C++
```

```
~unique_ptr();
```

Remarks

The destructor calls `get_deleter()(stored_ptr)`.

`weak_ptr` Class

Article • 06/21/2022

Wraps a weakly linked pointer.

Syntax

```
C++
```

```
template<class T> class weak_ptr;
```

Parameters

T

The type controlled by the weak pointer.

Remarks

The class template describes an object that points to a resource that is managed by one or more shared_ptr objects. The `weak_ptr` objects that point to a resource don't affect the resource's reference count. When the last `shared_ptr` object that manages that resource is destroyed, the resource will be freed, even if there are `weak_ptr` objects pointing to that resource. This behavior is essential for avoiding cycles in data structures.

A `weak_ptr` object points to a resource if it was constructed from a `shared_ptr` object that owns that resource, if it was constructed from a `weak_ptr` object that points to that resource, or if that resource was assigned to it with operator=. A `weak_ptr` object doesn't provide direct access to the resource that it points to. Code that needs to use the resource does so through a `shared_ptr` object that owns that resource, created by calling the member function lock. A `weak_ptr` object has expired when the resource that it points at has been freed because all of the `shared_ptr` objects that own the resource have been destroyed. Calling `lock` on a `weak_ptr` object that has expired creates an empty `shared_ptr` object.

An empty `weak_ptr` object doesn't point to any resources and has no control block. Its member function `lock` returns an empty `shared_ptr` object.

A cycle occurs when two or more resources controlled by `shared_ptr` objects hold mutually referencing `shared_ptr` objects. For example, a circular linked list with three elements has a head node `N0`; that node holds a `shared_ptr` object that owns the next node, `N1`; that node holds a `shared_ptr` object that owns the next node, `N2`; that node, in turn, holds a `shared_ptr` object that owns the head node, `N0`, closing the cycle. In this situation, the reference counts never become zero, and the nodes in the cycle are never freed. To eliminate the cycle, the last node `N2` should hold a `weak_ptr` object pointing to `N0` instead of a `shared_ptr` object. Since the `weak_ptr` object doesn't own `N0` it doesn't affect `N0`'s reference count, and when the program's last reference to the head node is destroyed the nodes in the list will also be destroyed.

Members

Name	Description
Constructors	
weak_ptr	Constructs a `weak_ptr`.
Destructors	
~weak_ptr	Destroys a `weak_ptr`.
Typedefs	
element_type	The type of the element.
Member functions	
expired	Tests if ownership has expired.
lock	Obtains exclusive ownership of a resource.
owner_before	Returns **`true`** if this `weak_ptr` is ordered before (or less than) the provided pointer.
reset	Releases owned resource.
swap	Swaps two `weak_ptr` objects.
use_count	Counts number of `shared_ptr` objects.
Operators	
operator=	Replaces owned resource.

`element_type`

The type of the element.

C++

```
typedef T element_type; // through C++17
using element_type = remove_extent_t<T>; // C++20
```

Remarks

The type is a synonym for the template parameter `T`.

Example

C++

```
// std__memory__weak_ptr_element_type.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::weak_ptr<int> wp0(sp0);
    std::weak_ptr<int>::element_type val = *wp0.lock();

    std::cout << "*wp0.lock() == " << val << std::endl;

    return (0);
}
```

Output

```
*wp0.lock() == 5
```

`expired`

Tests if ownership has expired, that is, the referenced object has been deleted.

C++

```
bool expired() const noexcept;
```

Remarks

The member function returns `true` if `*this` has expired, otherwise `false`.

Example

C++

```
// std__memory__weak_ptr_expired.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int(10));
        wp = sp;
        std::cout << "wp.expired() == " << std::boolalpha
            << wp.expired() << std::endl;
        std::cout << "*wp.lock() == " << *wp.lock() << std::endl;
    }

    // check expired after sp is destroyed
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;
    std::cout << "(bool)wp.lock() == " << std::boolalpha
        << (bool)wp.lock() << std::endl;

    return (0);
}
```

Output

```
wp.expired() == false
*wp.lock() == 10
wp.expired() == true
(bool)wp.lock() == false
```

`lock`

Obtains a `shared_ptr` that shares ownership of a resource.

C++

```
shared_ptr<T> lock() const noexcept;
```

Remarks

The member function returns an empty shared_ptr object if `*this` has expired; otherwise it returns a `shared_ptr<T>` object that owns the resource that `*this` points to. Returns a value equivalent to the atomic execution of `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`.

Example

C++

```
// std__memory__weak_ptr_lock.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp;

    {
        std::shared_ptr<int> sp(new int(10));
        wp = sp;
        std::cout << "wp.expired() == " << std::boolalpha
            << wp.expired() << std::endl;
        std::cout << "*wp.lock() == " << *wp.lock() << std::endl;
    }

    // check expired after sp is destroyed
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;
    std::cout << "(bool)wp.lock() == " << std::boolalpha
        << (bool)wp.lock() << std::endl;

    return (0);
}
```

Output

```
wp.expired() == false
*wp.lock() == 10
wp.expired() == true
(bool)wp.lock() == false
```

`operator=`

Replaces owned resource.

C++

```
weak_ptr& operator=(const weak_ptr& ptr) noexcept;

template <class Other>
weak_ptr& operator=(const weak_ptr<Other>& ptr) noexcept;

template <class Other>
weak_ptr& operator=(const shared_ptr<Other>& ptr) noexcept;
```

Parameters

`Other`

The type controlled by the argument shared or weak pointer.

`ptr`

The weak pointer or shared pointer to copy.

Remarks

The operators all release the resource currently pointed to by **`*this`** and assign ownership of the resource named by *`ptr`* to **`*this`**. If an operator fails, it leaves **`*this`** unchanged. Each operator has an effect equivalent to `weak_ptr(ptr).swap(*this)`.

Example

C++

```
// std__memory__weak_ptr_operator_as.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp0(new int(5));
    std::weak_ptr<int> wp0(sp0);
    std::cout << "*wp0.lock() == " << *wp0.lock() << std::endl;

    std::shared_ptr<int> sp1(new int(10));
    wp0 = sp1;
```

```
    std::cout << "*wp0.lock() == " << *wp0.lock() << std::endl;

    std::weak_ptr<int> wp1;
    wp1 = wp0;
    std::cout << "*wp1.lock() == " << *wp1.lock() << std::endl;

    return (0);
}
```

Output

```
*wp0.lock() == 5
*wp0.lock() == 10
*wp1.lock() == 10
```

`owner_before`

Returns **`true`** if this `weak_ptr` is ordered before (or less than) the provided pointer.

C++

```
template <class Other>
bool owner_before(const shared_ptr<Other>& ptr) const noexcept;

template <class Other>
bool owner_before(const weak_ptr<Other>& ptr) const noexcept;
```

Parameters

`ptr`

An `lvalue` reference to either a `shared_ptr` or a `weak_ptr`.

Remarks

The template member function returns **`true`** if **`*this`** is ordered before *`ptr`*.

`reset`

Releases the owned resource.

C++

```
void reset() noexcept;
```

Remarks

The member function releases the resource pointed to by `*this` and converts `*this` to an empty `weak_ptr` object.

Example

C++

```
// std__memory__weak_ptr_reset.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp(new int(5));
    std::weak_ptr<int> wp(sp);
    std::cout << "*wp.lock() == " << *wp.lock() << std::endl;
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;

    wp.reset();
    std::cout << "wp.expired() == " << std::boolalpha
        << wp.expired() << std::endl;

    return (0);
}
```

Output

```
*wp.lock() == 5
wp.expired() == false
wp.expired() == true
```

swap

Swaps two `weak_ptr` objects.

C++

```
void swap(weak_ptr& wp) noexcept;
```

Also includes the specialization:

C++

```
template<class T>
void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
```

Parameters

wp

The weak pointer to swap with.

Remarks

After a `swap`, the resource originally pointed to by `*this` is pointed to by *wp*, and the resource originally pointed to by *wp* is pointed to by `*this`. The function doesn't change the reference counts for the two resources and it doesn't throw any exceptions. The effect of the template specialization is the equivalent of `a.swap(b)`.

Example

C++

```
// std__memory__weak_ptr_swap.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::shared_ptr<int> sp2(new int(10));
    std::cout << "*sp1 == " << *sp1 << std::endl;

    sp1.swap(sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;

    swap(sp1, sp2);
    std::cout << "*sp1 == " << *sp1 << std::endl;
    std::cout << std::endl;

    std::weak_ptr<int> wp1(sp1);
    std::weak_ptr<int> wp2(sp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    wp1.swap(wp2);
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    swap(wp1, wp2);
```

```
    std::cout << "*wp1 == " << *wp1.lock() << std::endl;

    return (0);
}
```

Output

```
*sp1 == 5
*sp1 == 10
*sp1 == 5

*wp1 == 5
*wp1 == 10
*wp1 == 5
```

use_count

Counts the number of `shared_ptr` objects that own the shared resource.

C++

```
long use_count() const noexcept;
```

Remarks

The member function returns the number of `shared_ptr` objects that own the resource pointed to by `*this`.

Example

C++

```
// std__memory__weak_ptr_use_count.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::shared_ptr<int> sp1(new int(5));
    std::weak_ptr<int> wp(sp1);
    std::cout << "wp.use_count() == "
        << wp.use_count() << std::endl;

    std::shared_ptr<int> sp2(sp1);
```

```
    std::cout << "wp.use_count() == "
        << wp.use_count() << std::endl;

    return (0);
}
```

Output

```
wp.use_count() == 1
wp.use_count() == 2
```

weak_ptr

Constructs a `weak_ptr`.

C++

```
constexpr weak_ptr() noexcept;

weak_ptr(const weak_ptr& wp) noexcept;

weak_ptr(weak_ptr&& wp) noexcept;

template <class Other>
weak_ptr(const weak_ptr<Other>& wp) noexcept;

template <class Other>
weak_ptr(weak_ptr<Other>&& sp) noexcept;

template <class Other>
weak_ptr(const shared_ptr<Other>& sp) noexcept;
```

Parameters

`Other`

The type controlled by the argument shared/weak pointer. These constructors don't participate in overload resolution unless `Other*` is compatible with `element_type*`.

`wp`

The weak pointer to copy.

`sp`

The shared pointer to copy.

Remarks

The default constructor constructs an empty `weak_ptr` object. The constructors that take an argument each construct an empty `weak_ptr` object if the argument pointer is empty. Otherwise, they construct a `weak_ptr` object that points to the resource named by the argument. The reference count of the shared object isn't changed.

Example

C++

```
// std__memory__weak_ptr_construct.cpp
// compile with: /EHsc
#include <memory>
#include <iostream>

int main()
{
    std::weak_ptr<int> wp0;
    std::cout << "wp0.expired() == " << std::boolalpha
        << wp0.expired() << std::endl;

    std::shared_ptr<int> sp1(new int(5));
    std::weak_ptr<int> wp1(sp1);
    std::cout << "*wp1.lock() == "
        << *wp1.lock() << std::endl;

    std::weak_ptr<int> wp2(wp1);
    std::cout << "*wp2.lock() == "
        << *wp2.lock() << std::endl;

    return (0);
}
```

Output

```
wp0.expired() == true
*wp1.lock() == 5
*wp2.lock() == 5
```

~weak_ptr

Destroys a `weak_ptr`.

C++

```
~weak_ptr();
```

Remarks

The destructor destroys this `weak_ptr` but has no effect on the reference count of the object its stored pointer points at.

See also

Header Files Reference
<memory>
shared_ptr class

<memory_resource>

Article • 12/07/2021

Defines the container class template memory_resource and its supporting templates.

Syntax

```
C++
```

```
#include <memory_resource>
```

Members

Operators

Name	Description
operator!=	Tests if the memory_resource object on the left side of the operator is not equal to the memory_resource object on the right side.
operator==	Tests if the memory_resource object on the left side of the operator is equal to the memory_resource object on the right side.

Specialized Template Functions

Name	Description
polymorphic_allocator	

Functions

Name	Description
get_default_resource	
new_delete_resource	
null_memory_resource	
set_default_resource	

Classes and Structs

Name	Description
memory_resource Class	
monotonic_buffer_resource Class	
pool_options Struct	
synchronized_pool_resource Class	
unsynchronized_pool_resource Class	

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<memory_resource>` functions

Article • 12/07/2021

get_default_resource

C++

```
memory_resource* get_default_resource() noexcept;
```

new_delete_resource

C++

```
memory_resource* new_delete_resource() noexcept;
```

null_memory_resource

C++

```
memory_resource* null_memory_resource() noexcept;
```

polymorphic_allocator

C++

```
template <class Tp>
class polymorphic_allocator {
    memory_resource* memory_rsrc; // exposition only

    public:
    using value_type = Tp;

    polymorphic_allocator() noexcept;
    polymorphic_allocator(memory_resource* r);
    polymorphic_allocator(const polymorphic_allocator& other) = default;
    template <class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other)
noexcept;
    polymorphic_allocator&
        operator=(const polymorphic_allocator& rhs) = delete;
```

```
    Tp* allocate(size_t n);
    void deallocate(Tp* p, size_t n);
    template <class T, class... Args>
        void construct(T* p, Args&&... args);
    template <class T1, class T2, class... Args1, class... Args2>
        void construct(pair<T1,T2>* p, piecewise_construct_t,
    tuple<Args1...> x, tuple<Args2...> y);
    template <class T1, class T2>
        void construct(pair<T1,T2>* p);
    template <class T1, class T2, class U, class V>
        void construct(pair<T1,T2>* p, U&& x, V&& y);
    template <class T1, class T2, class U, class V>
        void construct(pair<T1,T2>* p, const pair<U, V>& pr);
    template <class T1, class T2, class U, class V>
        void construct(pair<T1,T2>* p, pair<U, V>&& pr);
    template <class T>
        void destroy(T* p);
    polymorphic_allocator select_on_container_copy_construction() const;
    memory_resource* resource() const;
};
```

set_default_resource

C++

```
memory_resource* set_default_resource(memory_resource* r) noexcept;
```

`<memory_resource>` operators

Article • 12/07/2021

operator!=

Tests if the memory_resource object on the left side of the operator is not equal to the memory_resource object on the right side.

C++

```
template <class T1, class T2>
    bool operator!=(const polymorphic_allocator<T1>& a, const
polymorphic_allocator<T2>& b) noexcept;
```

operator==

Tests if the memory_resource object on the left side of the operator is equal to the memory_resource object on the right side.

C++

```
template <class T1, class T2>
    bool operator==(const polymorphic_allocator<T1>& a, const
polymorphic_allocator<T2>& b) noexcept;
```

memory_resource Class

Article • 08/03/2021

Syntax

C++

```
class memory_resource {
    static constexpr size_t max_align = alignof(max_align_t); // exposition
only

    virtual ~memory_resource();
    void* allocate(size_t bytes, size_t alignment = max_align);
    void deallocate(void* p, size_t bytes, size_t alignment = max_align);
    bool is_equal(const memory_resource& other) const noexcept;
};
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`<mutex>`

Article • 03/01/2022

Include the standard header `<mutex>` to define the classes `mutex`, `recursive_mutex`, `timed_mutex`, and `recursive_timed_mutex`; the templates `lock_guard` and `unique_lock`; and supporting types and functions that define mutual-exclusion code regions.

> ⚠ **Warning**
>
> Beginning in Visual Studio 2015, the C++ Standard Library synchronization types are based on Windows synchronization primitives and no longer use ConcRT (except when the target platform is Windows XP). The types defined in `<mutex>` should not be used with any ConcRT types or functions.

Requirements

Header: `<mutex>`

Namespace: `std`

Remarks

> ⓘ **Note**
>
> In code that is compiled by using `/clr`, this header is blocked.

The classes `mutex` and `recursive_mutex` are *mutex types*. A mutex type has a default constructor and a destructor that doesn't throw exceptions. These objects have methods that provide mutual exclusion when multiple threads try to lock the same object. Specifically, a mutex type contains the methods `lock`, `try_lock`, and `unlock`:

- The `lock` method blocks the calling thread until the thread obtains ownership of the mutex. Its return value is ignored.
- The `try_lock` method tries to obtain ownership of the mutex without blocking. Its return type is convertible to `bool` and is `true` if the method obtains ownership, but is otherwise `false`.

- The `unlock` method releases the ownership of the mutex from the calling thread.

You can use mutex types as type arguments to instantiate the templates `lock_guard` and `unique_lock`. You can use objects of these types as the `Lock` argument to the wait member functions in the template condition_variable_any.

A *timed mutex type* satisfies the requirements for a mutex type. In addition, it has the `try_lock_for` and `try_lock_until` methods that must be callable by using one argument and must return a type that is convertible to **`bool`**. A timed mutex type can define these functions by using additional arguments, provided that those additional arguments all have default values.

- The `try_lock_for` method must be callable by using one argument, `Rel_time`, whose type is an instantiation of chrono::duration. The method tries to obtain ownership of the mutex, but returns within the time that is designated by `Rel_time`, regardless of success. The return value converts to **`true`** if the method obtains ownership; otherwise, the return value converts to **`false`**.

- The `try_lock_until` method must be callable by using one argument, `Abs_time`, whose type is an instantiation of chrono::time_point. The method tries to obtain ownership of the mutex, but returns no later than the time that is designated by `Abs_time`, regardless of success. The return value converts to **`true`** if the method obtains ownership; otherwise, the return value converts to **`false`**.

A mutex type is also known as a *lockable type*. If it doesn't provide the member function `try_lock`, it's a *basic lockable type*. A timed mutex type is known as a *timed lockable type*. A *timed lockable type* blocks for a specified amount of time, or until a lock on the mutex is acquired.

Members

Classes

Name	Description
lock_guard Class	Represents a template that can be instantiated to create an object whose destructor unlocks a mutex.
mutex Class (C++ Standard Library)	Represents a mutex type. Use objects of this type to enforce mutual exclusion within a program.

Name	Description
recursive_mutex Class	Represents a mutex type. In contrast to the `mutex` class, the behavior of calling locking methods for objects that are already locked is well-defined.
recursive_timed_mutex Class	Use objects of this type to enforce mutual exclusion that has time-limited blocking within a program. Unlike `timed_mutex`, the effect of calling locking methods for `recursive_timed_mutex` objects is well-defined.
scoped_lock Class	
timed_mutex Class	Use objects of this type to enforce mutual exclusion that has time-limited blocking within a program.
unique_lock Class	Represents a template that can be instantiated to create objects that manage the locking and unlocking of a mutex.

Functions

Name	Description
call_once	Provides a mechanism for calling a specified callable object exactly once during execution.
lock	Attempts to lock all arguments without deadlock.
swap	Tries to exchange the internal states of two `mutex` objects, `x` and `y`.
try_lock	Tries to lock the lockable object. Returns immediately.

Structs

Name	Description
adopt_lock_t Structure	Represents a type that is used to define an `adopt_lock`.
defer_lock_t Structure	Represents a type that defines a `defer_lock` object that is used to select one of the overloaded constructors of `unique_lock`.
once_flag Structure	Represents a `struct` that is used with the template function `call_once` to ensure that initialization code is called only once, even in the presence of multiple threads of execution.

Name	Description
try_to_lock_t Structure	Represents a `struct` that defines a `try_to_lock` object and is used to select one of the overloaded constructors of `unique_lock`.

Variables

Name	Description
adopt_lock	Represents an object that can be passed to constructors for `lock_guard` and `unique_lock` to indicate that the `mutex` object that is also being passed to the constructor is locked.
defer_lock	Represents an object that can be passed to the constructor for `unique_lock`, to indicate that the constructor shouldn't lock the `mutex` object that is also being passed to it.
try_to_lock	Represents an object that can be passed to the constructor for `unique_lock` to indicate that the constructor should try to unlock the `mutex` object that is also being passed to it without blocking.

See also

Header Files Reference

`<mutex>` functions and variables

Article • 12/07/2021

adopt_lock

Represents an object that can be passed to constructors for lock_guard and unique_lock to indicate that the mutex object that is also being passed to the constructor is locked.

```
C++

const adopt_lock_t adopt_lock;
```

call_once

Provides a mechanism for calling a specified callable object exactly once during execution.

```
C++

template <class Callable, class... Args>
void call_once(once_flag& Flag,
    Callable F&&, Args&&... A);
```

Parameters

Flag
A once_flag object that ensures that the callable object is only called once.

F
A callable object.

A
An argument list.

Remarks

If *Flag* is not valid, the function throws a system_error that has an error code of `invalid_argument`. Otherwise, the template function uses its *Flag* argument to ensure that it calls `F(A...)` successfully exactly once, regardless of how many times the

template function is called. If `F(A...)` exits by throwing an exception, the call was not successful.

defer_lock

Represents an object that can be passed to the constructor for unique_lock. This indicates that the constructor should not lock the mutex object that's also being passed to it.

C++

```
const defer_lock_t defer_lock;
```

lock

Attempts to lock all arguments without deadlock.

C++

```
template <class L1, class L2, class... L3>
void lock(L1&, L2&, L3&...);
```

Remarks

The arguments to the template function must be *mutex types*, except that calls to `try_lock` might throw exceptions.

The function locks all of its arguments without deadlock by calls to `lock`, `try_lock`, and `unlock`. If a call to `lock` or `try_lock` throws an exception, the function calls `unlock` on any of the mutex objects that were successfully locked before rethrowing the exception.

swap

C++

```
template <class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

try_lock

C++

```
template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

try_to_lock

Represents an object that can be passed to the constructor for unique_lock to indicate that the constructor should try to unlock the `mutex` that is also being passed to it without blocking.

C++

```
const try_to_lock_t try_to_lock;
```

adopt_lock_t Structure

Article • 08/03/2021

Represents a type that is used to define an adopt_lock.

Syntax

```
C++
```

```
struct adopt_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

Header Files Reference

<mutex>

defer_lock_t Structure

Article • 08/03/2021

Represents a type that defines a defer_lock object that is used to select one of the overloaded constructors of unique_lock.

Syntax

```
C++

struct defer_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

Header Files Reference
<mutex>

lock_guard Class

Article • 08/03/2021

Represents a template that can be instantiated to create an object whose destructor unlocks a `mutex`.

Syntax

```
C++
```

```
template <class Mutex>
class lock_guard;
```

Remarks

The template argument `Mutex` must name a *mutex type*.

Members

Public Typedefs

Name	Description
`lock_guard::mutex_type`	Synonym for the template argument `Mutex`.

Public Constructors

Name	Description
lock_guard	Constructs a `lock_guard` object.
lock_guard::~lock_guard Destructor	Unlocks the `mutex` that was passed to the constructor.

Requirements

Header: <mutex>

Namespace: std

lock_guard::lock_guard Constructor

Constructs a `lock_guard` object.

```
C++
```

```
explicit lock_guard(mutex_type& Mtx);

lock_guard(mutex_type& Mtx, adopt_lock_t);
```

Parameters

Mtx
A *mutex type* object.

Remarks

The first constructor constructs an object of type `lock_guard` and locks *Mtx*. If *Mtx* is not a recursive mutex, it must be unlocked when this constructor is called.

The second constructor does not lock *Mtx*. *Mtx* must be locked when this constructor is called. The constructor throws no exceptions.

lock_guard::~lock_guard Destructor

Unlocks the `mutex` that was passed to the constructor.

```
C++
```

```
~lock_guard() noexcept;
```

Remarks

If the `mutex` does not exist when the destructor runs, the behavior is undefined.

See also

Header Files Reference
<mutex>

`mutex` class (C++ Standard Library)

Article • 06/13/2024

Represents a *mutex type*. Objects of this type can be used to enforce mutual exclusion within a program.

Syntax

```
C++

class mutex;
```

Members

Public constructor/destructor

Expand table

Name	Description
mutex	Constructs a `mutex` object.
~mutex	Releases any resources that were used by the `mutex` object.

Public methods

Expand table

Name	Description
lock	Blocks the calling thread until the thread obtains ownership of the `mutex`.
native_handle	Returns the implementation-specific type that represents the mutex handle.
try_lock	Attempts to obtain ownership of the `mutex` without blocking.
unlock	Releases ownership of the `mutex`.

Requirements

Header: `<mutex>`

Namespace: `std`

`lock`

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
C++
```

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

Constructor

Constructs a `mutex` object that isn't locked.
Before Visual Studio 2022 17.10, Microsoft's implementation of this constructor wasn't `constexpr`. Now it's `constexpr`.

```
C++
```

```
mutex() noexcept;
```

Destructor

Releases any resources that are used by the `mutex` object.

```
C++
```

```
~mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

`native_handle`

Returns the implementation-specific type that represents the mutex handle. The mutex handle can be used in implementation-specific ways.

```
C++

native_handle_type native_handle();
```

Return Value

`native_handle_type` is defined as a `Concurrency::critical_section *`. It's cast as `void *`.

`try_lock`

Attempts to obtain ownership of the `mutex` without blocking.

```
C++

bool try_lock();
```

Return value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, **`false`**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

`unlock`

Releases ownership of the `mutex`.

```
C++

void unlock();
```

Remarks

If the calling thread doesn't own the `mutex`, the behavior is undefined.

See also

Header Files Reference
<mutex>

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

once_flag Structure

Article • 10/20/2023

Represents a `struct` that is used with the template function call_once to ensure that initialization code is called only once, even in the presence of multiple threads of execution.

Syntax

C++

```
struct once_flag
{
   constexpr once_flag() noexcept;
};
```

Remarks

The `once_flag` `struct` has only a default constructor.

Objects of type `once_flag` can be created, but they cannot be copied.

Requirements

Header: <mutex>

Namespace: std

See also

Header Files Reference
<mutex>

recursive_mutex Class

Article • 08/03/2021

Represents a *mutex type*. In contrast to mutex, the behavior of calls to locking methods for objects that are already locked is well-defined.

Syntax

```
C++

class recursive_mutex;
```

Members

Public Constructors

Name	Description
recursive_mutex	Constructs a `recursive_mutex` object.
~recursive_mutex Destructor	Releases any resources that are used by the `recursive_mutex` object.

Public Methods

Name	Description
lock	Blocks the calling thread until the thread obtains ownership of the mutex.
try_lock	Attempts to obtain ownership of the mutex without blocking.
unlock	Releases ownership of the mutex.

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
C++

void lock();
```

Remarks

If the calling thread already owns the `mutex`, the method returns immediately, and the previous lock remains in effect.

recursive_mutex

Constructs a `recursive_mutex` object that is not locked.

```
C++

recursive_mutex();
```

~recursive_mutex

Releases any resources that are used by the object.

```
C++

~recursive_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
C++

bool try_lock() noexcept;
```

Return Value

`true` if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex**; otherwise, **false`.

Remarks

If the calling thread already owns the `mutex`, the function immediately returns `true`, and the previous lock remains in effect.

unlock

Releases ownership of the mutex.

```
C++
```

```
void unlock();
```

Remarks

This method releases ownership of the `mutex` only after it is called as many times as lock and try_lock have been called successfully on the `recursive_mutex` object.

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

Header Files Reference
<mutex>

recursive_timed_mutex Class

Article • 08/03/2021

Represents a *timed mutex type*. Objects of this type are used to enforce mutual exclusion by using time-limited blocking within a program. Unlike objects of type timed_mutex, the effect of calling locking methods for `recursive_timed_mutex` objects is well-defined.

Syntax

```
C++
```

```
class recursive_timed_mutex;
```

Members

Public Constructors

Name	Description
recursive_timed_mutex	Constructs a `recursive_timed_mutex` object that's not locked.
~recursive_timed_mutex Destructor	Releases any resources that are used by the `recursive_timed_mutex` object.

Public Methods

Name	Description
lock	Blocks the calling thread until the thread obtains ownership of the `mutex`.
try_lock	Attempts to obtain ownership of the `mutex` without blocking.
try_lock_for	Attempts to obtain ownership of the `mutex` for a specified time interval.
try_lock_until	Attempts to obtain ownership of the `mutex` until a specified time.
unlock	Releases ownership of the `mutex`.

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
C++
```

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the method returns immediately, and the previous lock remains in effect.

recursive_timed_mutex Constructor

Constructs a `recursive_timed_mutex` object that is not locked.

```
C++
```

```
recursive_timed_mutex();
```

~recursive_timed_mutex Destructor

Releases any resources that are used by the `recursive_timed_mutex` object.

```
C++
```

```
~recursive_timed_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
C++
```

```
bool try_lock() noexcept;
```

Return Value

`true` if the method successfully obtained ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, `false`.

Remarks

If the calling thread already owns the `mutex`, the function immediately returns `true`, and the previous lock remains in effect.

try_lock_for

Attempts to obtain ownership of the `mutex` without blocking.

```
C++
```

```
template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time
A chrono::duration object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, `false`.

Remarks

If the calling thread already owns the `mutex`, the method immediately returns **`true`**, and the previous lock remains in effect.

try_lock_until

Attempts to obtain ownership of the `mutex` without blocking.

```
C++
```

```
template <class Clock, class Duration>
bool try_lock_for(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time
A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex` or if the calling thread already owns the `mutex`; otherwise, **`false`**.

Remarks

If the calling thread already owns the `mutex`, the method immediately returns **`true`**, and the previous lock remains in effect.

unlock

Releases ownership of the `mutex`.

```
C++
```

```
void unlock();
```

Remarks

This method releases ownership of the `mutex` only after it is called as many times as lock, try_lock, try_lock_for, and try_lock_until have been called successfully on the `recursive_timed_mutex` object.

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

Header Files Reference
<mutex>

timed_mutex Class

Article • 08/03/2021

Represents a *timed mutex type*. Objects of this type are used to enforce mutual exclusion through time-limited blocking within a program.

Syntax

```
C++
```

```
class timed_mutex;
```

Members

Public Constructors

Name	Description
timed_mutex	Constructs a `timed_mutex` object that's not locked.
timed_mutex::~timed_mutex Destructor	Releases any resources that are used by the `timed_mutex` object.

Public Methods

Name	Description
lock	Blocks the calling thread until the thread obtains ownership of the `mutex`.
try_lock	Attempts to obtain ownership of the `mutex` without blocking.
try_lock_for	Attempts to obtain ownership of the `mutex` for a specified time interval.
try_lock_until	Attempts to obtain ownership of the `mutex` until a specified time.
unlock	Releases ownership of the `mutex`.

Requirements

Header: <mutex>

Namespace: std

timed_mutex::lock

Blocks the calling thread until the thread obtains ownership of the `mutex`.

```
C++
```

```
void lock();
```

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::timed_mutex Constructor

Constructs a `timed_mutex` object that is not locked.

```
C++
```

```
timed_mutex();
```

timed_mutex::~timed_mutex Destructor

Releases any resources that are used by the `mutex` object.

```
C++
```

```
~timed_mutex();
```

Remarks

If the object is locked when the destructor runs, the behavior is undefined.

timed_mutex::try_lock

Attempts to obtain ownership of the `mutex` without blocking.

```
C++
```

```
bool try_lock();
```

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, `false`.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::try_lock_for

Attempts to obtain ownership of the `mutex` without blocking.

C++

```
template <class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time
A chrono::duration object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, `false`.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::try_lock_until

Attempts to obtain ownership of the `mutex` without blocking.

C++

```
template <class Clock, class Duration>
bool try_lock_for(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time
A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, **`false`**.

Remarks

If the calling thread already owns the `mutex`, the behavior is undefined.

timed_mutex::unlock

Releases ownership of the `mutex`.

C++

```
void unlock();
```

Remarks

If the calling thread does not own the `mutex`, the behavior is undefined.

See also

Header Files Reference
<mutex>

try_to_lock_t Structure

Article • 08/03/2021

Represents a `struct` that defines a try_to_lock object. Used to select one of the overloaded constructors of unique_lock.

Syntax

```
C++

struct try_to_lock_t;
```

Requirements

Header: <mutex>

Namespace: std

See also

Header Files Reference
<mutex>

unique_lock Class

Article • 06/21/2022

Represents a template that can be instantiated to create objects that manage the locking and unlocking of a `mutex`.

Syntax

```
C++
```

```
template <class Mutex>
class unique_lock;
```

Remarks

The template argument `Mutex` must name a *mutex type*.

Internally, a `unique_lock` stores a pointer to an associated `mutex` object and a `bool` that indicates whether the current thread owns the `mutex`.

Members

Public Typedefs

Name	Description
`mutex_type`	Synonym for the template argument `Mutex`.

Public Constructors

Name	Description
unique_lock	Constructs a `unique_lock` object.
~unique_lock Destructor	Releases any resources that are associated with the `unique_lock` object.

Public Methods

Name	Description
lock	Blocks the calling thread until the thread obtains ownership of the associated `mutex`.
mutex	Retrieves the stored pointer to the associated `mutex`.
owns_lock	Specifies whether the calling thread owns the associated `mutex`.
release	Disassociates the `unique_lock` object from the associated `mutex` object.
swap	Swaps the associated `mutex` and ownership status with that of a specified object.
try_lock	Attempts to obtain ownership of the associated `mutex` without blocking.
try_lock_for	Attempts to obtain ownership of the associated `mutex` without blocking.
try_lock_until	Attempts to obtain ownership of the associated `mutex` without blocking.
unlock	Releases ownership of the associated `mutex`.

Public Operators

Name	Description
operator bool	Specifies whether the calling thread has ownership of the associated `mutex`.
operator=	Copies the stored `mutex` pointer and associated ownership status from a specified object.

Inheritance Hierarchy

unique_lock

Requirements

Header: <mutex>

Namespace: std

lock

Blocks the calling thread until the thread obtains ownership of the associated `mutex`.

```
C++

void lock();
```

Remarks

If the stored `mutex` pointer is NULL, this method throws a system_error that has an error code of `operation_not_permitted`.

If the calling thread already owns the associated `mutex`, this method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

Otherwise, this method calls `lock` on the associated `mutex` and sets the internal thread ownership flag to **`true`**.

mutex

Retrieves the stored pointer to the associated `mutex`.

```
C++

mutex_type *mutex() const noexcept;
```

operator bool

Specifies whether the calling thread has ownership of the associated mutex.

```
C++

explicit operator bool() noexcept
```

Return Value

`true` if the thread owns the mutex; otherwise **`false`**.

operator=

Copies the stored `mutex` pointer and associated ownership status from a specified object.

```
C++
```

```
unique_lock& operator=(unique_lock&& Other) noexcept;
```

Parameters

Other
A `unique_lock` object.

Return Value

`*this`

Remarks

If the calling thread owns the previously associated `mutex`, before this method calls `unlock` on the `mutex`, it assigns the new values.

After the copy, this method sets *Other* to a default-constructed state.

owns_lock

Specifies whether the calling thread owns the associated `mutex`.

```
C++
```

```
bool owns_lock() const noexcept;
```

Return Value

`true` if the thread owns the `mutex`; otherwise, **`false`**.

release

Disassociates the `unique_lock` object from the associated `mutex` object.

```
C++
```

```
mutex_type *release() noexcept;
```

Return Value

The previous value of the stored `mutex` pointer.

Remarks

This method sets the value of the stored `mutex` pointer to 0 and sets the internal `mutex` ownership flag to **`false`**.

swap

Swaps the associated `mutex` and ownership status with that of a specified object.

```
C++
```

```
void swap(unique_lock& Other) noexcept;
```

Parameters

Other
A `unique_lock` object.

try_lock

Attempts to obtain ownership of the associated `mutex` without blocking.

```
C++
```

```
bool try_lock() noexcept;
```

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, **`false`**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a system_error that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

try_lock_for

Attempts to obtain ownership of the associated `mutex` without blocking.

```
C++
```

```
template <class Rep, class Period>
bool try_lock_for(
    const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time
A chrono::duration object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, **`false`**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a system_error that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

try_lock_until

Attempts to obtain ownership of the associated `mutex` without blocking.

```
C++
```

```
template <class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& Abs_time);

bool try_lock_until(const xtime* Abs_time);
```

Parameters

Abs_time
A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Return Value

`true` if the method successfully obtains ownership of the `mutex`; otherwise, **`false`**.

Remarks

If the stored `mutex` pointer is NULL, the method throws a system_error that has an error code of `operation_not_permitted`.

If the calling thread already owns the `mutex`, the method throws a `system_error` that has an error code of `resource_deadlock_would_occur`.

unique_lock Constructor

Constructs a `unique_lock` object.

C++

```
unique_lock() noexcept;
unique_lock(unique_lock&& Other) noexcept;
explicit unique_lock(mutex_type& Mtx);

unique_lock(mutex_type& Mtx, adopt_lock_t Adopt);

unique_lock(mutex_type& Mtx, defer_lock_t Defer) noexcept;
unique_lock(mutex_type& Mtx, try_to_lock_t Try);

template <class Rep, class Period>
unique_lock(mutex_type& Mtx,
    const chrono::duration<Rep, Period>
Rel_time);

template <class Clock, class Duration>
unique_lock(mutex_type& Mtx,
    const chrono::time_point<Clock, Duration>
Abs_time);

unique_lock(mutex_type& Mtx,
    const xtime* Abs_time) noexcept;
```

Parameters

Mtx
A mutex type object.

Rel_time
A chrono::duration object that specifies the maximum amount of time that the method attempts to obtain ownership of the `mutex`.

Abs_time
A point in time that specifies the threshold after which the method no longer attempts to obtain ownership of the `mutex`.

Other
A `unique_lock` object.

Remarks

The first constructor constructs an object that has an associated mutex pointer value of 0.

The second constructor moves the associated mutex status from *Other*. After the move, *Other* is no longer associated with a mutex.

The remaining constructors store & *Mtx* as the stored `mutex` pointer. Ownership of the `mutex` is determined by the second argument, if it exists.

Name	Description
`No argument`	Ownership is obtained by calling the `lock` method on the associated `mutex` object.
`Adopt`	Ownership is assumed. `Mtx` must be locked when the constructor is called.
`Defer`	The calling thread is assumed not to own the `mutex` object. `Mtx` must not be locked when the constructor is called.
`Try`	Ownership is determined by calling `try_lock` on the associated `mutex` object. The constructor throws nothing.
`Rel_time`	Ownership is determined by calling `try_lock_for(Rel_time)`.
`Abs_time`	Ownership is determined by calling `try_lock_until(Abs_time)`.

~unique_lock Destructor

Releases any resources that are associated with the `unique_lock` object.

```
C++
```

```
~unique_lock() noexcept;
```

Remarks

If the calling thread owns the associated `mutex`, the destructor releases ownership by calling unlock on the `mutex` object.

unlock

Releases ownership of the associated `mutex`.

```
C++
```

```
void unlock();
```

Remarks

If the calling thread doesn't own the associated `mutex`, this method throws a system_error that has an error code of `operation_not_permitted`.

Otherwise, this method calls `unlock` on the associated `mutex` and sets the internal thread ownership flag to **`false`**.

See also

Header Files Reference
<mutex>

<new>

Article • 12/07/2021

Defines several types and functions that control the allocation and freeing of storage under program control. It also defines components for reporting on storage management errors.

Requirements

Header: <new>

Namespace: std

Remarks

Some of the functions declared in this header are replaceable. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

Members

Objects

Name	Description
nothrow	Provides an object to be used as an argument for the `nothrow` versions of `new` and `delete`.

Typedefs

Name	Description
new_handler	A type that points to a function suitable for use as a new handler.
hardware_constructive_interference_size	
hardware_destructive_interference_size	

Functions

Name	Description
get_new_handler	
launder	
set_new_handler	Installs a user function that is called when new fails in its attempt to allocate memory.

Operators

Name	Description
operator delete	The function called by a delete expression to deallocate storage for individual of objects.
operator delete[]	The function called by a delete expression to deallocate storage for an array of objects.
operator new	The function called by a new expression to allocate storage for individual objects.
operator new[]	The function called by a new expression to allocate storage for an array of objects.

Enums

Name	Description
align_val_t	

Classes

Name	Description
bad_alloc Class	The class describes an exception thrown to indicate that an allocation request did not succeed.
bad_array_new_length Class	
nothrow_t Class	The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

`<new>` functions

Article • 12/07/2021

get_new_handler

```
C++
```

```
new_handler get_new_handler() noexcept;
```

Remarks

Returns the current `new_handler`.

launder

```
C++
```

```
template <class T>
    constexpr T* launder(T* ptr) noexcept;
```

Parameters

ptr
The address of a byte in memory that holds an object whose type is similar to *T*.

Return Value

A value of type *T** that points to X.

Remarks

Also referred to as a pointer optimization barrier.

Used as a constant expression when the value of its argument may be used in a constant expression. A byte of storage is reachable through a pointer value that points to an object if within the storage occupied by another object, an object with a similar pointer.

Example

C++

```
struct X { const int n; };

X *p = new X{3};
const int a = p->n;
new (p) X{5}; // p does not point to new object because X::n is const
const int b = p->n; // undefined behavior
const int c = std::launder(p)->n; // OK
```

nothrow

Provides an object to be used as an argument for the `nothrow` versions of `new` and `delete`.

C++

```
extern const std::nothrow_t nothrow;
```

Remarks

The object is used as a function argument to match the parameter type std::nothrow_t.

Example

See operator new and operator new[] for examples of how `std::nothrow_t` is used as a function parameter.

set_new_handler

Installs a user function that is to be called when **operator new** fails in its attempt to allocate memory.

C++

```
new_handler set_new_handler(new_handler Pnew) throw();
```

Parameters

Pnew

The `new_handler` to be installed.

Return Value

0 on the first call and the previous `new_handler` on subsequent calls.

Remarks

The function stores *Pnew* in a static new handler pointer that it maintains, then returns the value previously stored in the pointer. The `new` handler is used by operator new.

Example

C++

```
// new_set_new_handler.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;
void __cdecl newhandler( )
{
   cout << "The new_handler is called:" << endl;
   throw bad_alloc( );
   return;
}

int main( )
{
   set_new_handler (newhandler);
   try
   {
      while ( 1 )
      {
         new int[5000000];
         cout << "Allocating 5000000 ints." << endl;
      }
   }
   catch ( exception e )
   {
      cout << e.what( ) << endl;
   }
}
```

Output

```
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
The new_handler is called:
bad allocation
```

`<new>` typedefs

Article • 05/30/2022

`hardware_constructive_interference_size`

C++

```
inline constexpr size_t hardware_constructive_interference_size =
implementation-defined;
```

Remarks

This number is the maximum recommended size of contiguous memory occupied by two objects accessed with temporal locality by concurrent threads. It must be at least `alignof(max_align_t)`.

Example

C++

```
struct together {
    atomic<int> dog;
    int puppy;
};

struct kennel {
    // Other data members...
    alignas(sizeof(together)) together pack;
    // Other data members...
};

static_assert(sizeof(together) <= hardware_constructive_interference_size);
```

`hardware_destructive_interference_size`

C++

```
inline constexpr size_t hardware_destructive_interference_size =
implementation-defined;
```

Remarks

This number is the minimum recommended offset between two concurrently accessed objects to avoid performance degradation due to contention introduced by the implementation. It must be at least `alignof(max_align_t)`.

Example

C++

```
struct keep_apart {
    alignas(hardware_destructive_interference_size) atomic<int> cat;
    alignas(hardware_destructive_interference_size) atomic<int> dog;
};
```

`new_handler`

The `new_handler` type points to a function suitable for use as a *`new handler`*, a function that can reclaim or release memory.

C++

```
typedef void (*new_handler)();
```

Remarks

When of function of this type is set by std::set_new_handler, the function is called by `operator new` or `operator new[]` when they can't satisfy a request for more storage.

Example

For an example that uses `new_handler` as a return value, see set_new_handler.

`<new>` operators and enums

Article • 05/30/2022

`enum align_val_t`

```
C++
```

```
enum class align_val_t : size_t {};
```

`operator delete`

The function called by a `delete` expression to deallocate storage for individual objects.

```
C++
```

```
void operator delete(void* ptr) noexcept;
void operator delete(void *, void*) noexcept;
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
```

Parameters

`ptr`
The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by a `delete` expression to render the value of *`ptr`* invalid. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *`ptr`* that's `nullptr` or that was returned by an earlier call to operator new.

The default behavior for a `nullptr` value in *`ptr`* is to do nothing. Any other value of *`ptr`* must be a value returned earlier by a `new` call as previously described. The default behavior for a non-null value of *`ptr`* is to reclaim storage allocated by the earlier call. It's unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new`, or to any of the `calloc`, `malloc`, or `realloc` functions.

The second function is called by a placement `delete` expression corresponding to a `new` expression of the form `new( std::size_t )`. It does nothing.

The third function is called by a placement `delete` expression corresponding to a `new` expression of the form `new( std::size_t, const std::nothrow_t& )`. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of `ptr` that is `nullptr` or that was returned by an earlier call to `operator new`. The default behavior is to evaluate `delete( ptr )`.

Example

See operator new for an example that uses `operator delete`.

`operator delete[]`

The function called by a `delete` expression to deallocate storage for an array of objects.

C++

```
void operator delete[](void* ptr) noexcept;
void operator delete[](void *, void*) noexcept;
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
```

Parameters

`ptr`
The pointer whose value is to be rendered invalid by the deletion.

Remarks

The first function is called by an `delete[]` expression to render the value of *`ptr`* invalid. The function is replaceable because the program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *`ptr`* that is `nullptr` or that was returned by an earlier call to operator new[]. The default behavior for a null value of *`ptr`* is to do nothing. Any other value of *`ptr`* must be a value returned earlier by a call as previously described. The default behavior for such a non-null value of *`ptr`* is to reclaim storage allocated by the earlier call. It's unspecified under what conditions part or all of such

reclaimed storage is allocated by a subsequent call to operator new, or to any of the `calloc`, `malloc`, or `realloc` functions.

The second function is called by a placement `delete[]` expression corresponding to a `new[]` expression of the form `new[]( std::size_t )`. It does nothing.

The third function is called by a placement `delete[]` expression corresponding to a `new[]` expression of the form `new[]( std::size_t, const std::nothrow_t& )`. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is to accept a value of *`ptr`* that is `nullptr` or that was returned by an earlier call to operator `new[]`. The default behavior is to evaluate `delete[]( ptr )`.

Example

See operator new[] for examples of the use of `operator delete[]`.

`operator new`

The function called by a new-expression to allocate storage for individual objects.

C++

```
void* operator new(std::size_t count);
void* operator new(std::size_t count, const std::nothrow_t&) noexcept;
void* operator new(std::size_t count, void* ptr) noexcept;
```

Parameters

`count`
The number of bytes of storage to be allocated.

`ptr`
The pointer to be returned.

Return value

A pointer to the lowest byte address of the newly allocated storage. Or *`ptr`*, if using the third form of the function.

Remarks

The first function is called by a `new` expression to allocate `count` bytes of storage suitably aligned to represent any object of that size. This function is *replaceable*. It means the program can define an alternate function with this function signature that replaces the default version defined by the C++ Standard Library.

The required behavior is to return a non-null pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (the lowest byte address) of the allocated storage. If *`count`* is zero, the value returned doesn't compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to `malloc` is unspecified. If the allocation attempt is successful, the function returns a pointer to the allocated storage. Otherwise, the function calls the designated function of type new_handler. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function doesn't return.

The required behavior of a function of type `new_handler` is to perform one of the following operations:

- Make more storage available for allocation and then return.
- Call either `abort` or `exit`.
- Throw an object of type `bad_alloc`.

The default behavior of a new_handler function is to throw an object of type `bad_alloc`. A `nullptr` value designates the default `new_handler` function.

The order and contiguity of storage allocated by successive calls to `operator new` is unspecified, as are the initial values stored there.

To free storage allocated by the first form of `operator new`, call operator delete.

The second function is called by a placement `new` expression to allocate *`count`* bytes of storage suitably aligned to represent any object of that size. This function is *replaceable*. It means the program can define an alternate function with this function signature that replaces the default version defined by the C++ Standard Library.

The default behavior is to return `operator new( count )` if that function succeeds. Otherwise, it returns `nullptr`.

To free storage allocated by the second form of `operator new` (that is, if it didn't return `nullptr`), call operator delete.

The third function is called by a non-allocating placement `new` expression, of the form `new ( ptr ) T`. Here, *`ptr`* consists of a single object pointer. It can be useful for constructing an object at a known address. The function returns *`ptr`*. You must call the destructor explicitly on this object.

If you call non-allocating placement `new`, don't call `delete`. Instead, call the deallocator for the memory you provided, if necessary, after you call the destructor for the object.

For information on throwing or non-throwing behavior of `new`, see The new and delete operators.

Example

C++

```
// new_op_new.cpp
// compile with: /EHsc
#include<new>
#include<iostream>

using namespace std;

class MyClass {
    int member{ 0 };
public:
    MyClass() {
        cout << "MyClass at 0x" << this << " constructed.\n";
    };

    ~MyClass() {
        cout << "MyClass at 0x" << this << " destroyed.\n";
    };
};

int main( )
{
    // The first form of new / delete
    MyClass* fPtr1 = new MyClass;
    delete fPtr1;

    // The second form (fail returns nullptr) of new / delete
    MyClass* fPtr2 = new(nothrow) MyClass[2];
    if (fPtr2)
```

```
        delete fPtr2;

    // The third form (non-allocating placement) of new / delete
    char x[sizeof(MyClass)]; // x is automatic
    MyClass* fPtr3 = new(&x[0]) MyClass;
    fPtr3->~MyClass(); // Requires explicit destructor call
    // no delete because x is on the stack
}
```

`operator new[]`

The allocation function called by a new expression to allocate storage for an array of objects.

C++

```
void* operator new[](std::size_t count);
void* operator new[](std::size_t count, const std::nothrow_t&) noexcept;
void* operator new[](std::size_t count, void* ptr) noexcept;
```

Parameters

`count`

The number of bytes of storage to be allocated for the array object.

`ptr`

The pointer to be returned.

Return value

A pointer to the lowest byte address of the newly allocated storage. Or *`ptr`*, when the third form is used.

Remarks

The first function is called by a `new[]` expression to allocate *`count`* bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The required behavior is the same as for operator new. The default behavior is to return `operator new( count )` if it succeeds. Otherwise, it throws a `std::bad_alloc` exception (or an exception derived from `std::bad_alloc`). To free storage allocated by this form of `operator new[]`, call operator delete[].

The second function is called by a placement `new[]` expression to allocate *`count`* bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces the default version defined by the C++ Standard Library. The default behavior is to return `operator new( count )` if that function succeeds. Otherwise, it returns `nullptr`. To free storage allocated by this form of `operator new[]`, call `operator delete[]`. For more information on the throwing or non-throwing behavior of `new`, see The new and delete operators.

The third function is called by a non-allocating placement `new[]` expression, of the form `new( ptr ) T[ N ]`. This form doesn't allocate memory. It constructs the objects in the memory passed in through the *`ptr`* parameter. The function returns *`ptr`*. You must call the destructor explicitly for each object created. You're responsible for providing sufficient memory for *`ptr`*. Don't invoke `delete[]` on the value returned by the `new` expression. Instead, deallocate *`ptr`*, if necessary, after you call the destructors.

Example

C++

```
// new_op_array.cpp
// compile with: /EHsc
#include <new>
#include <iostream>

using namespace std;

class MyClass {
    int member{ 0 };
public:
    MyClass() {
        cout << "MyClass at 0x" << this << " constructed.\n";
    };

    ~MyClass() {
        cout << "MyClass at 0x" << this << " destroyed.\n";
    };
};

int main() {
    // The first form of array new / delete
    MyClass* fPtr1 = new MyClass[2];
    delete[] fPtr1;

    // The second form (fail returns nullptr) of array new / delete
    MyClass* fPtr2 = new(nothrow) MyClass[2];
    if (fPtr2)
        delete[] fPtr2;
```

```
    // The third form (non-allocating placement) of array new / delete
    char x[2 * sizeof(MyClass) + sizeof(int)]; // x is automatic

    MyClass* fPtr3 = new(&x[0]) MyClass[2];
    fPtr3[1].~MyClass(); // Requires explicit destructor calls
    fPtr3[0].~MyClass(); // Recommended in reverse construction order
    // Don't delete[] fPtr3 here.
    // delete[] &x[0] not required because x is on the stack
}
```

bad_alloc Class

Article • 08/03/2021

The class describes an exception thrown to indicate that an allocation request did not succeed.

Syntax

C++

```
class bad_alloc : public exception {
    bad_alloc();
    virtual ~bad_alloc();
    bad_alloc(const bad_alloc&);
    bad_alloc& operator=(const bad_alloc&);
    const char* what() const override;
};
```

Remarks

The value returned by `what` is an implementation-defined C string. None of the member functions throw any exceptions.

Example

C++

```
// bad_alloc.cpp
// compile with: /EHsc
#include<new>
#include<iostream>
using namespace std;

int main() {
   char* ptr;
   try {
      ptr = new char[(~unsigned int((int)0)/2) - 1];
      delete[] ptr;
   }
   catch( bad_alloc &ba) {
      cout << ba.what( ) << endl;
   }
}
```

Output

```
bad allocation
```

nothrow_t Structure

Article • 11/11/2021

The struct is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

Syntax

```
C++
```

```
struct std::nothrow_t {};
```

Remarks

The struct helps the compiler to select the correct version of the constructor. nothrow is a synonym for objects of type `std::nothrow_t`.

Example

See operator new and operator new[] for examples of how `std::nothrow_t` is used as a function parameter.

<numeric>

Article • 12/07/2021

Defines container template functions that perform algorithms for numerical processing.

Requirements

Header: <numeric>

Namespace: std

Remarks

The numeric algorithms resemble the C++ Standard Library algorithms in <algorithm>, and can operate on a variety of data structures. These include standard library container classes—for example, vector and list, and program-defined data structures and arrays of elements that satisfy the requirements of a particular algorithm. The algorithms achieve this level of generality by accessing and traversing the elements of a container indirectly through iterators. The algorithms process iterator ranges that are typically specified by their beginning or ending positions. The ranges referred to must be valid in the sense that all pointers in the ranges must be dereferenceable and within the sequences of each range, and the last position must be reachable from the first by means of incrementation.

The algorithms extend the actions that are supported by the operations and member functions of each of the C++ Standard Library containers and enable interaction with different types of container objects at the same time.

Members

Functions

Name	Description
accumulate	Computes the sum of all elements in a specified range—including some initial value—by computing successive partial sums, or computes the result of successive partial results that are obtained by using a specified binary operation instead of the sum operation.

Name	Description
adjacent_difference	Computes the successive differences between each element and its predecessor in an input range and outputs the results to a destination range, or computes the result of a generalized procedure where the difference operation is replaced by another specified binary operation.
exclusive_scan	
gcd	
inclusive_scan	
inner_product	Computes the sum of the element-wise product of two ranges and adds it to a specified initial value, or computes the result of a generalized procedure where the sum and product operations are replaced by other specified binary operations.
iota	Stores a starting value, beginning with the first element and filling with successive increments of the value (`value++`) in each of the elements in the interval `[first, last)`.
lcm	
partial_sum	Computes a series of sums in an input range from the first element through the *i*th element and stores the result of each sum in the *i*th element of a destination range, or computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.
reduce	
transform_exclusive_scan	
transform_inclusive_scan	
transform_reduce	

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<numeric>` functions

Article • 12/07/2021

accumulate

Computes the sum of all the elements in a specified range, including some initial value, by computing successive partial sums. Or, computes the result of successive partial results of a specified binary operation.

C++

```
template <class InputIterator, class Type>
Type accumulate(
    InputIterator first,
    InputIterator last,
    Type init);

template <class InputIterator, class Type, class BinaryOperation>
Type accumulate(
    InputIterator first,
    InputIterator last,
    Type init,
    BinaryOperation binary_op);
```

Parameters

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

Return value

The sum of *init* and all the elements in the specified range for the first template function, or, for the second template function, the result of applying the binary operation *binary_op* instead of the sum operation, to (*PartialResult, *in_iter*), where *PartialResult* is the result of previous applications of the operation and *in_iter* is an iterator pointing to the next element in the range.

Remarks

The initial value ensures that there's a well-defined result when the range is empty, in which case *init* is returned. The binary operation doesn't need to be associative or commutative. The result is initialized to the initial value *init* and then *result* = *binary_op*(*result*, *in_iter*) is calculated iteratively through the range, where *in_iter* is an iterator pointing to each successive element in the range. The range must be valid, and the complexity is linear with the size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

C++

```
// numeric_accum.cpp
// compile with: /EHsc
#include <vector>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2(20);
   vector <int>::iterator iter1, iter2;

   int i;
   for (i = 1; i < 21; i++)
   {
      v1.push_back(i);
   }

   cout << "The original vector v1 is:\n ( " ;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
      cout << *iter1 << " ";
   cout << ")." << endl;
```

```
    // The first member function for the accumulated sum
    int total;
    total = accumulate(v1.begin(), v1.end(), 0);

    cout << "The sum of the integers from 1 to 20 is: "
         << total << "." << endl;

    // Constructing a vector of partial sums
    int j = 0, partotal;
    for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
    {
       partotal = accumulate(v1.begin(), iter1 + 1, 0);
       v2[j] = partotal;
       j++;
    }

    cout << "The vector of partial sums is:\n ( " ;
    for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
       cout << *iter2 << " ";
    cout << ")." << endl << endl;

    // The second member function for the accumulated product
    vector <int> v3, v4(10);
    vector <int>::iterator iter3, iter4;

    int s;
    for (s = 1; s < 11; s++)
    {
       v3.push_back(s);
    }

    cout << "The original vector v3 is:\n ( " ;
    for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
       cout << *iter3 << " ";
    cout << ")." << endl;

    int ptotal;
    ptotal = accumulate(v3.begin(), v3.end(), 1, multiplies<int>());

    cout << "The product of the integers from 1 to 10 is: "
         << ptotal << "." << endl;

    // Constructing a vector of partial products
    int k = 0, ppartotal;
    for (iter3 = v3.begin(); iter3 != v3.end(); iter3++) {
       ppartotal = accumulate(v3.begin(), iter3 + 1, 1, multiplies<int>());
       v4[k] = ppartotal;
       k++;
    }

    cout << "The vector of partial products is:\n ( " ;
    for (iter4 = v4.begin(); iter4 != v4.end(); iter4++)
       cout << *iter4 << " ";
    cout << ")." << endl;
}
```

Output

```
The original vector v1 is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The sum of the integers from 1 to 20 is: 210.
The vector of partial sums is:
( 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 ).

The original vector v3 is:
( 1 2 3 4 5 6 7 8 9 10 ).
The product of the integers from 1 to 10 is: 3628800.
The vector of partial products is:
( 1 2 6 24 120 720 5040 40320 362880 3628800 ).
```

adjacent_difference

Computes the successive differences between each element and its predecessor in an input range. Outputs the results to a destination range. Or, computes the result of a generalized procedure where the difference operation is replaced by another, specified binary operation.

C++

```
template <class InputIterator, class OutIterator>
OutputIterator adjacent_difference(
    InputIterator first,
    InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutIterator, class BinaryOperation>
OutputIterator adjacent_difference(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

template <class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 adjacent_difference(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result);

template <class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2,
class BinaryOperation>
ForwardIterator2 adjacent_difference(
```

```
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    BinaryOperation binary_op);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

last
An input iterator addressing the last element in the input range whose elements are to be differenced with their respective predecessors or where the pair of values is to be operated on by another specified binary operation.

result
An output iterator addressing the first element a destination range where the series of differences or the results of the specified operation is to be stored.

binary_op
The binary operation to apply in the generalized operation, replacing the operation of subtraction in the differencing procedure.

Return value

An output iterator addressing the end of the destination range: `result` + (`last` - `first`).

Remarks

The output iterator *result* is allowed to be the same iterator as the input iterator *first*, so that `adjacent_difference` values may be computed in place.

For a sequence of values *a*1, *a*2, *a*3, in an input range, the first template function stores successive `adjacent_difference` values *a*1, *a*2 - *a*1, a3 - *a*2, in the destination range.

For a sequence of values *a*1, *a*2, *a*3, in an input range, the second template function stores successive `adjacent_difference` values *a*1, *a*2 *binary_op* *a*1, *a*3 *binary_op* *a*2, in the destination range.

The binary operation *binary_op* isn't required to be either associative or commutative, because the order of operations applied is specified.

Example

C++

```
// numeric_adj_diff.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
   using namespace std;

   vector<int> V1( 10 ), V2( 10 );
   vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

   list <int> L1;
   list <int>::iterator LIter1, LIterend, LIterend2;

   int t;
   for ( t = 1 ; t <= 10 ; t++ )
   {
      L1.push_back( t * t );
   }

   cout << "The input list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
      cout << *LIter1 << " ";
   cout << ")." << endl;

   // The first member function for the adjacent_differences of
   // elements in a list output to a vector
   VIterend = adjacent_difference ( L1.begin ( ) , L1.end ( ) ,
      V1.begin ( ) );

   cout << "Output vector containing adjacent_differences is:\n ( " ;
   for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
      cout << *VIter1 << " ";
   cout << ")." << endl;

   // The second member function used to compute
```

```
   // the adjacent products of the elements in a list
   VIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , V2.begin (
) ,
      multiplies<int>( ) );

   cout << "The output vector with the adjacent products is:\n ( " ;
   for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
      cout << *VIter2 << " ";
   cout << ")." << endl;

   // Computation of adjacent_differences in place
   LIterend2 = adjacent_difference ( L1.begin ( ) , L1.end ( ) , L1.begin (
) );
   cout << "In place output adjacent_differences in list L1 is:\n ( " ;
   for ( LIter1 = L1.begin( ) ; LIter1 != LIterend2 ; LIter1++ )
      cout << *LIter1 << " ";
   cout << ")." << endl;
}
```

exclusive_scan

Computes an exclusive prefix sum operation by using either `std::plus<>()` or a specified binary operator over a range, given an initial value. Writes the results to the range beginning at the specified destination. An *exclusive prefix* sum means the *n*th input element isn't included in the *n*th sum. Overloads that include an execution policy argument execute according to the specified policy.

C++

```
template<class InputIterator, class OutputIterator, class Type>
OutputIterator exclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    Type init);

template<class InputIterator, class OutputIterator, class Type, class
BinaryOperation>
OutputIterator exclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    Type init,
    BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type>
ForwardIterator2 exclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
```

```
    ForwardIterator1 last,
    ForwardIterator2 result,
    Type init);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type, class BinaryOperation>
ForwardIterator2 exclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    Type init,
    BinaryOperation binary_op);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

Return value

An output iterator addressing the end of the destination range: *result* + (*last* - *first*).

gcd

Computes the greatest common divisor of the integers m and n.

```
C++
```

```
template <class M, class N>
constexpr common_type_t<M,N> gcd(M m, N n);
```

Parameters

m, *n*
Values of integral type.

Return value

Returns the greatest common divisor of the absolute values of *m* and *n*, or zero if both *m* and *n* are zero. The results are undefined if the absolute values of *m* or *n* aren't representable as values of type `common_type_t<M,N>`.

inclusive_scan

Computes an inclusive prefix sum operation by using either `std::plus<>()` or a specified binary operator over a range, given an initial value. Writes the results to the range beginning at the specified destination. An *inclusive prefix* sum means the *n*th input element is included in the *n*th sum. Overloads that include an execution policy argument execute according to the specified policy.

```
C++
```

```
template<class InputIterator, class OutputIterator>
OutputIterator inclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator inclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation,
class Type>
OutputIterator inclusive_scan(
    InputIterator first,
```

```
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op,
    Type init);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2>
ForwardIterator2 inclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2,
class BinaryOperation>
ForwardIterator2 inclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2,
class BinaryOperation, class Type>
ForwardIterator2 inclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    BinaryOperation binary_op,
    Type init);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

Return value

An output iterator addressing the end of the destination range: *result* + (*last* - *first*).

inner_product

Computes the sum of the element-wise product of two ranges and adds it to a specified initial value or computes the result of a generalized procedure where the sum and product binary operations are replaced by other specified binary operations.

C++

```
template <class InputIterator1, class InputIterator2, class Type>
Type inner_product(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    Type             init);

template <class InputIterator1, class InputIterator2, class Type, class
BinaryOperation1, class BinaryOperation2>
Type inner_product(
    InputIterator1   first1,
    InputIterator1   last1,
    InputIterator2   first2,
    Type             init,
    BinaryOperation1 binary_op1,
    BinaryOperation2 binary_op2);
```

Parameters

first1
An input iterator addressing the first element in the first range whose inner product or generalized inner product with the second range is to be computed.

last1
An input iterator addressing the last element in the first range whose inner product or generalized inner product with the second range is to be computed.

first2
An input iterator addressing the first element in the second range whose inner product or generalized inner product with the first range is to be computed.

init
An initial value to which the inner product or generalized inner product between the ranges is to be added.

binary_op1
The binary operation that replaces the inner product operation of sum applied to the element-wise products in the generalization of the inner product.

binary_op2
The binary operation that replaces the inner product element-wise operation of multiply in the generalization of the inner product.

Return value

The first member function returns the sum of the element-wise products and adds to it the specified initial value. So for ranges of values *a*i and *b*i, it returns:

init + (*a*1 * *b*1) + (*a*2 * *b*2) + ... + (*a*n * *b*n)

by iteratively replacing *init* with *init* + (*a*i * *b*i).

The second member function returns:

init binary_op1 (*a*1 *binary_op2 b*1) *binary_op1* (*a*2 *binary_op2 b*2) *binary_op1* ... *binary_op1* (*a*n *binary_op2 b*n)

by iteratively replacing *init* with *init binary_op1* (*a*i *binary_op2 b*i).

Remarks

The initial value ensures that there's a well-defined result when the range is empty. In that case, *init* is returned. The binary operations don't need to be associative or commutative. The range must be valid, and the complexity is linear with the size of the range. The return type of the binary operator must be convertible to **Type** to ensure closure during the iteration.

Example

C++

```
// numeric_inner_prod.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main()
{
   using namespace std;

   vector <int> v1, v2(7), v3(7);
   vector <int>::iterator iter1, iter2, iter3;

   int i;
   for (i = 1; i <= 7; i++)
   {
      v1.push_back(i);
   }

   cout << "The original vector v1 is:\n ( " ;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
      cout << *iter1 << " ";
   cout << ")." << endl;

   list <int> l1, l2(7);
   list <int>::iterator lIter1, lIter2;

   int t;
   for (t = 1; t <= 7; t++)
   {
      l1.push_back(t);
   }

   cout << "The original list l1 is:\n ( " ;
   for (lIter1 = l1.begin(); lIter1 != l1.end(); lIter1++)
      cout << *lIter1 << " ";
   cout << ")." << endl;

   // The first member function for the inner product
   int inprod;
   inprod = inner_product(v1.begin(), v1.end(), l1.begin(), 0);

   cout << "The inner_product of the vector v1 and the list l1 is: "
        << inprod << "." << endl;

   // Constructing a vector of partial inner_products between v1 & l1
   int j = 0, parinprod;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++) {
```

```
      parinprod = inner_product(v1.begin(), iter1 + 1, l1.begin(), 0);
      v2[j] = parinprod;
      j++;
   }

   cout << "Vector of partial inner_products between v1 & l1 is:\n ( " ;
   for (iter2 = v2.begin(); iter2 != v2.end(); iter2++)
      cout << *iter2 << " ";
   cout << ")." << endl << endl;

   // The second member function used to compute
   // the product of the element-wise sums
   int inprod2;
   inprod2 = inner_product (v1.begin(), v1.end(),
      l1.begin(), 1, multiplies<int>(), plus<int>());

   cout << "The sum of the element-wise products of v1 and l1 is: "
        << inprod2 << "." << endl;

   // Constructing a vector of partial sums of element-wise products
   int k = 0, parinprod2;
   for (iter1 = v1.begin(); iter1 != v1.end(); iter1++)
   {
      parinprod2 =
         inner_product(v1.begin(), iter1 + 1, l1.begin(), 1,
         multiplies<int>(), plus<int>());
      v3[k] = parinprod2;
      k++;
   }

   cout << "Vector of partial sums of element-wise products is:\n ( " ;
   for (iter3 = v3.begin(); iter3 != v3.end(); iter3++)
      cout << *iter3 << " ";
   cout << ")." << endl << endl;
}
```

iota

Stores a starting value, beginning with the first element and filling with successive increments of that value (`value++`) in each of the elements in the interval `[first, last)`.

C++

```
template <class ForwardIterator, class Type>
void iota(ForwardIterator first, ForwardIterator last, Type value);
```

Parameters

first
An input iterator that addresses the first element in the range to be filled.

last
An input iterator that addresses the last element in the range to be filled.

value
The starting value to store in the first element and to successively increment for later elements.

Example

The following example demonstrates some uses for the `iota` function by filling a list of integers and then filling a vector with the `list` so that the random_shuffle function can be used.

C++

```
// compile by using: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <numeric>
#include <list>
#include <vector>
#include <iostream>

using namespace std;

int main(void)
{
    list <int> intList(10);
    vector <list<int>::iterator> intVec(intList.size());

    // Fill the list
    iota(intList.begin(), intList.end(), 0);

    // Fill the vector with the list so we can shuffle it
    iota(intVec.begin(), intVec.end(), intList.begin());

    random_shuffle(intVec.begin(), intVec.end());

    // Output results
    cout << "Contents of the integer list: " << endl;
    for (auto i: intList) {
        cout << i << ' ';
    }
    cout << endl << endl;

    cout << "Contents of the integer list, shuffled by using a vector: " <<
endl;
    for (auto i: intVec) {
```

```
        cout << *i << ' ';
    }
    cout << endl;
}
```

lcm

C++

```
template <class M, class N>
constexpr common_type_t<M,N> lcm(M m, N n);
```

partial_sum

Computes a series of sums in an input range from the first element through the *n*th element and stores the result of each such sum in the *n*th element of a destination range. Or, computes the result of a generalized procedure where the sum operation is replaced by another specified binary operation.

C++

```
template <class InputIterator, class OutIt>
OutputIterator partial_sum(
    InputIterator first,
    InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutIt, class Fn2>
OutputIterator partial_sum(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op);
```

Parameters

first
An input iterator addressing the first element in the range to be partially summed or combined according to a specified binary operation.

last
An input iterator addressing the last element in the range to be partially summed or

combined according to a specified binary operation that is one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element of a destination range to store the series of partial sums, or the successive results of the specified binary operation.

binary_op
The binary operation to apply in the generalized operation, replacing the operation of sum in the partial sum procedure.

Return value

An output iterator addressing the end of the destination range: *result* + (*last* - *first*).

Remarks

The output iterator *result* is allowed to be the same iterator as the input iterator *first*, so that partial sums may be computed in place.

For a sequence of values *a*1, *a*2, ... *a*x, in an input range, the first template function stores successive partial sums in the destination range. The *n*th element is given by (*a*1 + *a*2 + *a*3 + ... + *a*n).

For a sequence of values *a*1, *a*2, *a*3, in an input range, the second template function stores successive partial results in the destination range. The *n*th element is given by ((... ((*a*1 *binary_op a*2) *binary_op a*3) *binary_op* ...) *binary_op a*n).

The binary operation *binary_op* isn't required to be either associative or commutative, because the order of operations applied is specified.

Example

C++

```
// numeric_partial_sum.cpp
// compile with: /EHsc
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>

int main( )
{
```

```
    using namespace std;
    vector<int> V1( 10 ), V2( 10 );
    vector<int>::iterator VIter1, VIter2, VIterend, VIterend2;

    list <int> L1;
    list <int>::iterator LIter1, LIterend;

    int t;
    for ( t = 1 ; t <= 10 ; t++ )
    {
       L1.push_back( t );
    }

    cout << "The input list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != L1.end( ) ; LIter1++ )
       cout << *LIter1 << " ";
    cout << ")." << endl;

    // The first member function for the partial sums of
    // elements in a list output to a vector
    VIterend = partial_sum ( L1.begin ( ) , L1.end ( ) ,
       V1.begin ( ) );

    cout << "The output vector containing the partial sums is:\n ( " ;
    for ( VIter1 = V1.begin( ) ; VIter1 != VIterend ; VIter1++ )
       cout << *VIter1 << " ";
    cout << ")." << endl;

    // The second member function used to compute
    // the partial product of the elements in a list
    VIterend2 = partial_sum ( L1.begin ( ) , L1.end ( ) , V2.begin ( ) ,
       multiplies<int>( ) );

    cout << "The output vector with the partial products is:\n ( " ;
    for ( VIter2 = V2.begin( ) ; VIter2 != VIterend2 ; VIter2++ )
       cout << *VIter2 << " ";
    cout << ")." << endl;

    // Computation of partial sums in place
    LIterend = partial_sum ( L1.begin ( ) , L1.end ( ) , L1.begin ( ) );
    cout << "The in place output partial_sum list L1 is:\n ( " ;
    for ( LIter1 = L1.begin( ) ; LIter1 != LIterend ; LIter1++ )
       cout << *LIter1 << " ";
    cout << ")." << endl;
}
```

reduce

Reduces all the elements in a specified range, possibly including some initial value, by computing sums in an arbitrary and possibly permuted order. Or, reduces by computing

the results of a specified binary operation. Overloads that include an execution policy argument execute according to the specified policy.

```
C++
```

```
template<class InputIterator>
typename iterator_traits<InputIterator>::value_type reduce(
    InputIterator first,
    InputIterator last);

template<class InputIterator, class Type>
Type reduce(
    InputIterator first,
    InputIterator last,
    Type init);

template<class InputIterator, class Type, class BinaryOperation>
Type reduce(
    InputIterator first,
    InputIterator last,
    Type init,
    BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator>
typename iterator_traits<ForwardIterator>::value_type reduce(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Type>
Type reduce(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Type init);

template<class ExecutionPolicy, class ForwardIterator, class Type, class
BinaryOperation>
Type reduce(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Type init,
    BinaryOperation binary_op);
```

Parameters

exec

An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

Return value

The result of applying *binary_op* or `std::plus<>()` to *init* and all the elements in the specified range to (*PartialResult, *in_iter*), where *PartialResult* is the result of previous applications of the operation, and *in_iter* is an iterator pointing to some element in the range. In overloads that don't specify *init*, the *init* value used is equivalent to `typename iterator_traits<InputIterator>::value_type{}`.

Remarks

`reduce` behavior is non-deterministic unless *binary_op* is associative and commutative. The behavior is undefined if *binary_op* modifies any element, or invalidates any iterator in the interval [*first*, *last*], inclusive.

transform_exclusive_scan

Transforms the elements of a range with a specified unary operator, then computes an exclusive prefix sum operation by using either `std::plus<>()` or a specified binary operator over the range, given an initial value. Writes the results to the range beginning at the specified destination. An *exclusive prefix* sum means the *n*th input element isn't included in the *n*th sum. Overloads that include an execution policy argument execute

according to the specified policy. The summation may be performed in an arbitrary order.

```
C++
```

```
template<class InputIterator, class OutputIterator, class Type, class
BinaryOperation, class UnaryOperation>
OutputIterator transform_exclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    Type init,
    BinaryOperation binary_op,
    UnaryOperation unary_op);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type, class BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_exclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    Type init,
    BinaryOperation binary_op,
    UnaryOperation unary_op);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

unary_op
The unary operation to apply to each element in the specified range.

transform_inclusive_scan

Transforms the elements of a range with a specified unary operator, then computes an inclusive prefix sum operation by using either `std::plus<>()` or a specified binary operator over the range, given an initial value. Writes the results to the range beginning at the specified destination. An *inclusive prefix* sum means the *n*th input element is included in the *n*th sum. Overloads that include an execution policy argument execute according to the specified policy. The summation may be performed in an arbitrary order.

C++

```
template<class InputIterator, class OutputIterator, class BinaryOperation,
class UnaryOperation>
OutputIterator transform_inclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op,
    UnaryOperation unary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation,
class UnaryOperation, class Type>
OutputIterator transform_inclusive_scan(
    InputIterator first,
    InputIterator last,
    OutputIterator result,
    BinaryOperation binary_op,
    UnaryOperation unary_op,
    Type init);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_inclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    BinaryOperation binary_op,
    UnaryOperation unary_op);

template<class ExecutionPolicy, class ForwardIterator1, class
```

```
ForwardIterator2, class BinaryOperation, class UnaryOperation, class Type>
ForwardIterator2 transform_inclusive_scan(
    ExecutionPolicy&& exec,
    ForwardIterator1 first,
    ForwardIterator1 last,
    ForwardIterator2 result,
    BinaryOperation binary_op,
    UnaryOperation unary_op,
    Type init);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

unary_op
The unary operation to apply to each element in the specified range.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

transform_reduce

Transforms a range of elements, then applies a functor that reduces the transformed elements in arbitrary order. Effectively, a `transform` followed by a `reduce`.

```
C++
```

```
template<class InputIterator1, class InputIterator2, class Type>
Type transform_reduce(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    Type init);

template<class InputIterator1, class InputIterator2, class Type, class
BinaryOperation1, class BinaryOperation2>
Type transform_reduce(
    InputIterator1 first1,
    InputIterator1 last1,
    InputIterator2 first2,
    Type init,
    BinaryOperation1 binary_op1,
    BinaryOperation2 binary_op2);

template<class InputIterator, class Type, class BinaryOperation, class
UnaryOperation>
Type transform_reduce(
    InputIterator first,
    InputIterator last,
    Type init,
    BinaryOperation binary_op,
    UnaryOperation unary_op);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type>
Type transform_reduce(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    Type init);

template<class ExecutionPolicy, class ForwardIterator1, class
ForwardIterator2, class Type, class BinaryOperation1, class
BinaryOperation2>
Type transform_reduce(
    ExecutionPolicy&& exec,
    ForwardIterator1 first1,
    ForwardIterator1 last1,
    ForwardIterator2 first2,
    Type init,
    BinaryOperation1 binary_op1,
    BinaryOperation2 binary_op2);

template<class ExecutionPolicy, class ForwardIterator, class Type, class
BinaryOperation, class UnaryOperation>
Type transform_reduce(
    ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator last,
    Type init,
```

```
    BinaryOperation binary_op,
    UnaryOperation unary_op);
```

Parameters

exec
An execution policy.

first
An input iterator addressing the first element in the range to either sum or combine by using *binary_op*.

first1
An input iterator addressing the first element in the range to either sum or combine by using *binary_op1*.

last
An input iterator addressing the last element in the range to either sum or combine by using *binary_op*, that's one position beyond the final element actually included in the iterated accumulation.

last1
An input iterator addressing the last element in the range to either sum or combine by using *binary_op1*, that's one position beyond the final element actually included in the iterated accumulation.

result
An output iterator addressing the first element a destination range where the series of sums or the results of the specified operation is to be stored.

init
An initial value to which each element is in turn added or combined by using *binary_op*.

binary_op
The binary operation to apply to each element in the specified range and the result of its previous applications.

unary_op
The unary operation to apply to each element in the specified range.

Return value

The transformed then reduced result.

<optional>

Article • 12/07/2021

Defines the container class template `optional` and several supporting templates.

Requirements

Header: <optional>

Namespace: std

Members

Operators

Name	Description
operator==	Tests if an object is equal to another object.
operator!=	Tests if an object is not equal to another object.
operator<	Tests if the object on the left is less than the object on the right.
operator<=	Tests if the object on the left is less than or equal to the object on the right.
operator>	Tests if the object on the left is greater than the object on the right.
operator>=	Tests if the object on the left is greater than or equal to the object on the right.

> ⓘ **Note**
>
> In addition to relational compares, <optional> operators also support comparison with **nullopt** and `T`.

Functions

Name	Description
make_optional	Makes an object optional.
swap	Swaps the contained values of two `optional` objects.

Classes and Structs

Name	Description
hash	Returns a hash of the contained object.
optional class	Describes an object that may or may not hold a value.
nullopt_t struct	Describes an object not holding a value.
bad_optional_access class	Describes an object thrown as an exception to report an attempt to access a value not there.

Objects

Name	Description
nullopt	An instance of `nullopt_t` for comparisons.

See also

Header Files Reference

`<optional>` functions

Article • 12/07/2021

make_optional

Makes an object optional.

C++

```
template <class T>
    constexpr optional<see below> make_optional(T&&);
template <class T, class... Args>
    constexpr optional<T> make_optional(Args&&... args);
template <class T, class U, class... Args>
    constexpr optional<T> make_optional(initializer_list<U> il, Args&&...
args);
```

nullopt

C++

```
inline constexpr nullopt_t nullopt(unspecified );
```

swap

C++

```
template <class T>
    void swap(optional<T>&, optional<T>&) noexcept(see below );
```

`<optional>` operators

Article • 12/07/2021

operator==

Tests if the `optional` object on the left side of the operator is equal to the `optional` object on the right side.

C++

```
template <class T, class U> constexpr bool operator==(const optional<T>&
left, const optional<U>& right);
template <class T> constexpr bool operator==(const optional<T>& left,
nullopt_t right) noexcept;
template <class T> constexpr bool operator==(nullopt_t left, const
optional<T>& right) noexcept;
template <class T, class U> constexpr bool operator==(const optional<T>&,
const U&);
template <class T, class U> constexpr bool operator==(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

operator!=

Tests if the `optional` object on the left side of the operator is not equal to the `optional` object on the right side.

C++

```
template <class T, class U> constexpr bool operator!=(const optional<T>&,
const optional<U>&);
template <class T> constexpr bool operator!=(const optional<T>&, nullopt_t)
noexcept;
template <class T> constexpr bool operator!=(nullopt_t, const optional<T>&)
noexcept;
template <class T, class U> constexpr bool operator!=(const optional<T>&,
```

```
const U&);
template <class T, class U> constexpr bool operator!=(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

Remarks

This template function returns `!(left == right)`.

`operator<`

Tests if the `optional` object on the left side of the operator is less than the `optional` object on the right side.

C++

```
template <class T, class U> constexpr bool operator<(const optional<T>&,
const optional<U>&);
template <class T> constexpr bool operator<(const optional<T>&, nullopt_t)
noexcept;
template <class T> constexpr bool operator<(nullopt_t, const optional<T>&)
noexcept;
template <class T, class U> constexpr bool operator<(const optional<T>&,
const U&);
template <class T, class U> constexpr bool operator<(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

Return Value

`true` if the list on the left side of the operator is less than but not equal to the list on the right side of the operator; otherwise `false`.

`operator<=`

Tests if the `optional` object on the left side of the operator is less than or equal to the `optional` object on the right side.

```
C++
```

```
template <class T, class U> constexpr bool operator<=(const optional<T>&,
const optional<U>&);
template <class T> constexpr bool operator<=(const optional<T>&, nullopt_t)
noexcept;
template <class T> constexpr bool operator<=(nullopt_t, const optional<T>&)
noexcept;
template <class T, class U> constexpr bool operator<=(const optional<T>&,
const U&);
template <class T, class U> constexpr bool operator<=(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

Return Value

`true` if the list on the left side of the operator is less than or equal to the list on the right side of the operator; otherwise `false`.

Remarks

This template function returns `!(right < left)`.

`operator>`

Tests if the `optional` object on the left side of the operator is greater than the `optional` object on the right side.

```
C++
```

```
template <class T, class U> constexpr bool operator>(const optional<T>&,
const optional<U>&);
template <class T> constexpr bool operator>(const optional<T>&, nullopt_t)
noexcept;
template <class T> constexpr bool operator>(nullopt_t, const optional<T>&)
noexcept;
template <class T, class U> constexpr bool operator>(const optional<T>&,
const U&);
template <class T, class U> constexpr bool operator>(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

Return Value

`true` if the list on the left side of the operator is greater than the list on the right side of the operator; otherwise **`false`**.

Remarks

This template function returns `right < left`.

`operator>=`

Tests if the `optional` object on the left side of the operator is greater than or equal to the `optional` object on the right side.

```
C++
```

```
template <class T, class U> constexpr bool operator>=(const optional<T>&,
const optional<U>&);
template <class T> constexpr bool operator>=(const optional<T>&, nullopt_t)
```

```
noexcept;
template <class T> constexpr bool operator>=(nullopt_t, const optional<T>&)
noexcept;
template <class T, class U> constexpr bool operator>=(const optional<T>&,
const U&);
template <class T, class U> constexpr bool operator>=(const U&, const
optional<T>&);
```

Parameters

left
An object of type `optional`, `nullopt_t`, or `T`.

right
An object of type `optional`, `nullopt_t`, or `T`.

Return Value

`true` if the `optional` on the left side of the operator is greater than or equal to the `optional` on the right side of the operator; otherwise **`false`**.

Remarks

The template function returns `!(left < right)`.

optional Class

Article • 11/14/2024

The class template `optional<T>` describes an object that may or may not contain a value of type `T`, known as the *contained value*.

When an instance of `optional<T>` contains a value, the contained value is allocated within the storage of the `optional` object, in a region suitably aligned for type `T`. When an `optional<T>` is converted to `bool`, the result is `true` if the object contains a value; otherwise, it's `false`.

The contained object type `T` must not be in_place_t or nullopt_t. `T` must be *destructible*, that is, its destructor must reclaim all owned resources, and may throw no exceptions.

The `optional` class is new in C++17.

Syntax

C++

```
template <class T>
class optional
{
    using value_type = T;
};

template<class T> optional(T) -> optional<T>;
```

Members

Constructors

Expand table

Name	Description
Constructors and destructor	
optional	Constructs an object of type `optional`.

Name	Description
~optional	Destroys an object of type `optional`.
Assignment	
operator=	Replaces the `optional` with a copy of another `optional`.
emplace	Initializes the contained value with specified arguments.
Swap	
swap	Swaps the contained value or the empty state with another `optional`.
Observers	
has_value	Returns whether an `optional` object contains a value.
value	Returns the contained value.
value_or	Returns the contained value, or an alternative if no value is present.
operator->	References the contained value of an `optional` object.
operator*	References the contained value of an `optional` object.
operator bool	Returns whether an `optional` object contains a value.
Modifiers	
reset	Resets the `optional` by destroying any contained value.

has_value

C++

```
constexpr bool has_value() const noexcept;
```

optional constructor

Constructs an object of type `optional`.

C++

```
constexpr optional() noexcept;
constexpr optional(nullopt_t nullopt) noexcept;
constexpr optional(const optional& rhs);
```

```
constexpr optional(optional&& rhs) noexcept;

template <class... Args>
constexpr explicit optional(in_place_t, Args&&... args);

template <class U, class... Args>
constexpr explicit optional(in_place_t, initializer_list<U> i_list,
Args&&... args);

template <class U = T>
explicit constexpr optional(U&& rhs);

template <class U>
explicit optional(const optional<U>& rhs);

template <class U>
explicit optional(optional<U>&& rhs);
```

Parameters

rhs
The `optional` to copy or move construct the contained value from.

i_list
The initializer list to construct the contained value from.

args
The argument list to construct the contained value from.

Remarks

`constexpr optional() noexcept;` `constexpr optional(nullopt_t nullopt) noexcept;`
These constructors construct an `optional` that doesn't contain a value.

`constexpr optional(const optional& rhs);` The copy constructor initializes the contained value from the contained value of the argument. It's defined as **deleted** unless `is_copy_constructible_v<T>` is true, and it's trivial if `is_trivially_copy_constructible_v<T>` is true.

`constexpr optional(optional&& rhs) noexcept;` The move constructor initializes the contained value by moving from the contained value of the argument. It doesn't participate in overload resolution unless `is_move_constructible_v<T>` is true, and it's trivial if `is_trivially_move_constructible_v<T>` is true.

`template <class... Args> constexpr explicit optional(in_place_t, Args&&... args);` Direct initializes the contained value as if using the arguments `std::forward<Args>(args)`. This constructor is **`constexpr`** if the `T` constructor used is **`constexpr`**. It doesn't participate in overload resolution unless `is_constructible_v<T, Args...>` is true.

`template <class U, class... Args> constexpr explicit optional(in_place_t, initializer_list<U> i_list, Args&&... args);` Direct initializes the contained value as if using the arguments `i_list, std::forward<Args>(args)`. This constructor is **`constexpr`** if the `T` constructor used is **`constexpr`**. It doesn't participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is true.

`template <class U = T> explicit constexpr optional(U&& rhs);` Direct initializes the contained value as if using `std::forward<U>(v)`. This constructor is **`constexpr`** if the `T` constructor used is **`constexpr`**. It doesn't participate in overload resolution unless `is_constructible_v<T, U&&>` is true, and `is_same_v<remove_cvref_t<U>, in_place_t>` and `is_same_v<remove_cvref_t<U>, optional>` are false.

`template <class U> explicit optional(const optional<U>& rhs);` If *rhs* contains a value, direct initializes the contained value from the contained value of the argument. It doesn't participate in overload resolution unless `is_constructible_v<T, const U&>` is true, and `is_constructible_v<T, optional<U>&>`, `is_constructible_v<T, optional<U>&&>`, `is_constructible_v<T, const optional<U>&>`, `is_constructible_v<T, const optional<U>&&>`, `is_convertible_v<optional<U>&, T>`, `is_convertible_v<optional<U>&&, T>`, `is_convertible_v<const optional<U>&, T>`, and `is_convertible_v<const optional<U>&&, T>` are all false.

`template <class U> explicit optional(optional<U>&& rhs);` If *rhs* contains a value, direct initializes the contained value as if using `std::move(*rhs)`. It doesn't participate in overload resolution unless `is_constructible_v<T, U&&>` is true, and `is_constructible_v<T, optional<U>&>`, `is_constructible_v<T, optional<U>&&>`, `is_constructible_v<T, const optional<U>&>`, `is_constructible_v<T, const optional<U>&&>`, `is_convertible_v<optional<U>&, T>`, `is_convertible_v<optional<U>&&, T>`, `is_convertible_v<const optional<U>&, T>`, and `is_convertible_v<const optional<U>&&, T>` are all false.

~optional destructor

Destroys the contained value, if one is present.

```
C++
```

```
~optional();
```

Remarks

If `T` is trivially destructible, then `optional<T>` is also trivially destructible.

operator=

Replaces the contained value of an `optional` with a copy or move from another `optional` contained value.

C++

```
optional& operator=(nullopt_t) noexcept;
optional& operator=(const optional& rhs);
optional& operator=(optional&&) noexcept( /* see below */ );

template <class U = T>
    optional& operator=(U&&);

template <class U>
optional& operator=(const optional<U>&);

template <class U>
    optional& operator=(optional<U>&&);

template <class... Args>
T& emplace(Args&&...);

template <class U, class... Args>
T& emplace(initializer_list<U>, Args&&...);
```

operator->

Dereferences the contained value of an `optional` object.

C++

```
constexpr const T* operator->() const;
constexpr T* operator->();
```

operator*

Dereferences the contained value of an `optional` object.

```
C++

constexpr const T& operator*() const&;
constexpr T& operator*() &;
constexpr T&& operator*() &&;
constexpr const T&& operator*() const&&;
```

operator bool

Reports whether the `optional` object has a contained value.

```
C++

constexpr explicit operator bool() const noexcept;
```

reset

Effectively, calls the destructor of the contained object, if any, and sets it to an uninitialized state.

```
C++

void reset() noexcept;
```

swap

```
C++

template<class T>
void swap(optional<T>&, optional<T>&) noexcept;
```

value

```
C++

constexpr const T& value() const&;
constexpr T& value() &;
```

```
constexpr T&& value() &&;
constexpr const T&& value() const&&;
```

value_or

```
C++
```

```
template <class U>
    constexpr T value_or(U&&) const&;
template <class U>
    constexpr T value_or(U&&) &&;
```

See also

<optional>

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

<ostream>

Article • 12/07/2021

Defines the class template basic_ostream, which mediates insertions for the iostreams. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You rarely need to include it directly.)

Syntax

```
C++
```

```
#include <ostream>
```

Typedefs

Type name	Description
ostream	Creates a type from `basic_ostream` that is specialized on `char` and `char_traits` specialized on `char`.
wostream	Creates a type from `basic_ostream` that is specialized on `wchar_t` and `char_traits` specialized on `wchar_t`.

Manipulators

Name	Description
endl	Terminates a line and flushes the buffer.
ends	Terminates a string.
flush	Flushes the buffer.
swap	Exchanges the values of the left `basic_ostream` object parameter for those of the right `basic_ostream` object parameter.

Operators

Operator	Description

Operator	Description
operator<<	Writes various types to the stream.

Classes

Class	Description
basic_ostream	The class template describes an object that controls insertion of elements and encoded objects into a stream buffer.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

basic_ostream Class

Article • 12/07/2021

This class template describes an object that controls insertion of elements and encoded objects into a stream buffer with elements of type `Elem`, also known as char_type, whose character traits are determined by the class `Tr`, also known as traits_type.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_ostream : virtual public basic_ios<Elem, Tr>
```

Parameters

Elem
A `char_type`.

Tr
The character `traits_type`.

Remarks

Most of the member functions that overload operator<< are formatted output functions. They follow the pattern:

C++

```
iostate state = goodbit;
const sentry ok(*this);

if (ok)
{try
{<convert and insert elements
    accumulate flags in state> }
    catch (...)
{try
{setstate(badbit);

}
    catch (...)
{}
```

```
    if ((exceptions()& badbit) != 0)
    throw; }}
width(0);
// Except for operator<<(Elem)
setstate(state);

return (*this);
```

Two other member functions are unformatted output functions. They follow the pattern:

C++

```
iostate state = goodbit;
const sentry ok(*this);

if (!ok)
    state |= badbit;
else
{try
{<obtain and insert elements
    accumulate flags in state> }
    catch (...)
{try
{setstate(badbit);

}
    catch (...)
{}
    if ((exceptions()& badbit) != 0)
    throw; }}
setstate(state);

return (*this);
```

Both groups of functions call setstate(**badbit**) if they encounter a failure while inserting elements.

An object of class basic_istream< **Elem**, **Tr**> stores only a virtual public base object of class basic_ios<**Elem**, **Tr**>.

Example

See the example for basic_ofstream Class to learn more about output streams.

Constructors

Constructor	Description

Constructor	Description
basic_ostream	Constructs a `basic_ostream` object.

Member functions

Member function	Description
flush	Flushes the buffer.
put	Puts a character in a stream.
seekp	Resets position in output stream.
sentry	The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.
swap	Exchanges the values of this `basic_ostream` object for those of the provided `basic_ostream` object.
tellp	Reports position in output stream.
write	Puts characters in a stream.

Operators

Operator	Description
operator=	Assigns the value of the provided `basic_ostream` object parameter to this object.
operator<<	Writes to the stream.

Requirements

Header: <ostream>

Namespace: std

basic_ostream::basic_ostream

Constructs a `basic_ostream` object.

```
C++
```

```
explicit basic_ostream(
    basic_streambuf<Elem, Tr>* strbuf,
    bool _Isstd = false);

basic_ostream(basic_ostream&& right);
```

Parameters

strbuf
An object of type basic_streambuf.

_Isstd
`true` if this is a standard stream; otherwise, `false`.

right
An rvalue reference to an object of type `basic_ostream`.

Remarks

The first constructor initializes the base class by calling init(`strbuf`). The second constructor initializes the base class by calling basic_ios::move`(right)`.

Example

See the example for basic_ofstream::basic_ofstream to learn more about output streams.

basic_ostream::flush

Flushes the buffer.

```
C++

basic_ostream<Elem, Tr>& flush();
```

Return Value

A reference to the basic_ostream object.

Remarks

If rdbuf is not a null pointer, the function calls **rdbuf->**pubsync. If that returns -1, the function calls setstate(**badbit**). It returns ***this**.

Example

C++

```
// basic_ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "test";
   cout.flush();
}
```

Output

```
test
```

`basic_ostream::operator<<`

Writes to the stream.

C++

```
basic_ostream<Elem, Tr>& operator<<(
    basic_ostream<Elem, Tr>& (* Pfn)(basic_ostream<Elem, Tr>&));

basic_ostream<Elem, Tr>& operator<<(
    ios_base& (* Pfn)(ios_base&));

basic_ostream<Elem, Tr>& operator<<(
    basic_ios<Elem, Tr>& (* Pfn)(basic_ios<Elem, Tr>&));

basic_ostream<Elem, Tr>& operator<<(basic_streambuf<Elem, Tr>* strbuf);
basic_ostream<Elem, Tr>& operator<<(bool val);
basic_ostream<Elem, Tr>& operator<<(short val);
basic_ostream<Elem, Tr>& operator<<(unsigned short val);
basic_ostream<Elem, Tr>& operator<<(int __w64  val);
basic_ostream<Elem, Tr>& operator<<(unsigned int __w64  val);
basic_ostream<Elem, Tr>& operator<<(long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long __w64  val);
basic_ostream<Elem, Tr>& operator<<(long long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long long val);
basic_ostream<Elem, Tr>& operator<<(float val);
```

```
basic_ostream<Elem, Tr>& operator<<(double val);
basic_ostream<Elem, Tr>& operator<<(long double val);
basic_ostream<Elem, Tr>& operator<<(const void* val);
```

Parameters

Pfn
A function pointer.

strbuf
A pointer to a `stream_buf` object.

val
An element to write to the stream.

Return Value

A reference to the basic_ostream object.

Remarks

The <ostream> header also defines several global insertion operators. For more information, see operator<<.

The first member function ensures that an expression of the form `ostr << endl` calls endl(**ostr**), and then returns ***this**. The second and third functions ensure that other manipulators, such as hex, behave similarly. The remaining functions are all formatted output functions.

The function

```
C++

basic_ostream<Elem, Tr>& operator<<(basic_streambuf<Elem, Tr>* strbuf);
```

extracts elements from *strbuf*, if *strbuf* is not a null pointer, and inserts them. Extraction stops on end of file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls setstate(**failbit**). In any case, the function returns ***this**.

The function

C++

```
basic_ostream<Elem, Tr>& operator<<(bool val);
```

converts `_Val` to a Boolean field and inserts it by calling use_facet<**num_put<Elem, OutIt>**(getloc). put(**OutIt**(rdbuf), ***this**, `getloc`, **val**). Here, `OutIt` is defined as ostreambuf_iterator<**Elem, Tr**>. The function returns ***this**.

The functions

C++

```
basic_ostream<Elem, Tr>& operator<<(short val);
basic_ostream<Elem, Tr>& operator<<(unsigned short val);
basic_ostream<Elem, Tr>& operator<<(int val);
basic_ostream<Elem, Tr>& operator<<(unsigned int __w64  val);
basic_ostream<Elem, Tr>& operator<<(long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long val);
basic_ostream<Elem, Tr>& operator<<(long long val);
basic_ostream<Elem, Tr>& operator<<(unsigned long long val);
basic_ostream<Elem, Tr>& operator<<(const void* val);
```

each convert *val* to a numeric field and insert it by calling **use_facet<num_put<Elem, OutIt>**(`getloc`). **put**(**OutIt**(`rdbuf`), ***this**, `getloc`, **val**). Here, **OutIt** is defined as **ostreambuf_iterator<Elem, Tr>**. The function returns ***this**.

The functions

C++

```
basic_ostream<Elem, Tr>& operator<<(float val);
basic_ostream<Elem, Tr>& operator<<(double val);
basic_ostream<Elem, Tr>& operator<<(long double val);
```

each convert *val* to a numeric field and insert it by calling **use_facet<num_put<Elem, OutIt>**(`getloc`). **put**(**OutIt**(`rdbuf`), ***this**, `getloc`, **val**). Here, **OutIt** is defined as **ostreambuf_iterator<Elem, Tr>**. The function returns ***this**.

Example

C++

```
// basic_ostream_op_write.cpp
// compile with: /EHsc
#include <iostream>
```

```
#include <string.h>

using namespace std;

ios_base& hex2( ios_base& ib )
{
   ib.unsetf( ios_base::dec );
   ib.setf( ios_base::hex );
   return ib;
}

basic_ostream<char, char_traits<char> >& somefunc(basic_ostream<char,
char_traits<char> > &i)
{
    if (i == cout)
    {
        i << "i is cout" << endl;
    }
    return i;
}

class CTxtStreambuf : public basic_streambuf< char, char_traits< char > >
{
public:
    CTxtStreambuf(char *_pszText)
    {
        pszText = _pszText;
        setg(pszText, pszText, pszText + strlen(pszText));
    };
    char *pszText;
};

int main()
{
    cout << somefunc;
    cout << 21 << endl;

    hex2(cout);
    cout << 21 << endl;

    CTxtStreambuf f("text in streambuf");
    cout << &f << endl;
}
```

basic_ostream::operator=

Assigns values for the provided `basic_ostream` object parameter to this object.

C++

```
basic_ostream& operator=(basic_ostream&& right);
```

Parameters

right
An `rvalue` reference to a `basic_ostream` object.

Remarks

The member operator calls swap `(right)`.

basic_ostream::put

Puts a character in a stream.

```
C++
```

```
basic_ostream<Elem, Tr>& put(char_type _Ch);
```

Parameters

_Ch
A character.

Return Value

A reference to the basic_ostream object.

Remarks

The unformatted output function inserts the element *_Ch*. It returns ***this**.

Example

```
C++
```

```
// basic_ostream_put.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
```

```
    using namespace std;
    cout.put( 'v' );
    cout << endl;
    wcout.put( L'l' );
}
```

Output

```
v
l
```

basic_ostream::seekp

Reset position in output stream.

C++

```
basic_ostream<Elem, Tr>& seekp(pos_type _Pos);

basic_ostream<Elem, Tr>& seekp(off_type _Off, ios_base::seekdir _Way);
```

Parameters

_Pos
The position in the stream.

_Off
The offset relative to *_Way*.

_Way
One of the ios_base::seekdir enumerations.

Return Value

A reference to the basic_ostream object.

Remarks

If fail is `false`, the first member function calls **newpos** = rdbuf-> pubseekpos(*_Pos*), for some `pos_type` temporary object `newpos`. If `fail` is false, the second function calls **newpos** = **rdbuf->** pubseekoff(*_Off*, *_Way*). In either case, if (`off_type`)**newpos** ==

(`off_type`)(-1) (the positioning operation fails), then the function calls **istr**.setstate(**failbit**). Both functions return ***this**.

Example

C++

```
// basic_ostream_seekp.cpp
// compile with: /EHsc
#include <fstream>
#include <iostream>

int main()
{
    using namespace std;
    ofstream x("basic_ostream_seekp.txt");
    streamoff i = x.tellp();
    cout << i << endl;
    x << "testing";
    i = x.tellp();
    cout << i << endl;
    x.seekp(2);   // Put char in third char position in file
    x << " ";

    x.seekp(2, ios::end);   // Put char two after end of file
    x << "z";
}
```

Output

```
0
7
```

basic_ostream::sentry

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions.

class sentry { public: explicit sentry(basic_ostream<Elem, Tr>& _Ostr); operator bool() const; ~sentry(); };

Remarks

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions. If **ostr**.good is `true` and **ostr**.tie is not a

null pointer, the constructor calls **ostr.tie->**flush. The constructor then stores the value returned by `ostr.good` in `status`. A later call to `operator bool` delivers this stored value.

If `uncaught_exception` returns **`false`** and flags & unitbuf is nonzero, the destructor calls flush.

basic_ostream::swap

Exchanges the values of this `basic_ostream` object for the values of the provided `basic_ostream`.

```
C++
```

```
void swap(basic_ostream& right);
```

Parameters

right
A reference to a `basic_ostream` object.

Remarks

The member function calls basic_ios::swap`(right)` to exchange the contents of this object for the contents of *right*.

basic_ostream::tellp

Report position in output stream.

```
C++
```

```
pos_type tellp();
```

Return Value

Position in output stream.

Remarks

If fail is `false`, the member function returns rdbuf-> pubseekoff(0, `cur`, **in**). Otherwise, it returns `pos_type`(-1).

Example

See seekp for an example using `tellp`.

basic_ostream::write

Put characters in a stream.

```
C++
basic_ostream<Elem, Tr>& write(const char_type* str, streamsize count);
```

Parameters

count
Count of characters to put into the stream.

str
Characters to put into the stream.

Return Value

A reference to the basic_ostream object.

Remarks

The unformatted output function inserts the sequence of *count* elements beginning at *str*.

Example

See streamsize for an example using `write`.

See also

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

`<ostream>` functions

Article • 12/07/2021

These are the global template functions defined in `<ostream>`. For member functions, see the basic_ostream Class documentation.

endl
ends
flush
swap

endl

Terminates a line and flushes the buffer.

```
C++
```

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& endl(
   basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem
The element type.

Ostr
An object of type **basic_ostream**.

Tr
Character traits.

Return Value

An object of type **basic_ostream**.

Remarks

The manipulator calls *Ostr*.put(*Ostr*.widen('\n')), and then calls *Ostr*.flush. It returns *Ostr*.

Example

C++

```
// ostream_endl.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "testing" << endl;
}
```

Output

```
testing
```

ends

Terminates a string.

C++

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& ends(
   basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem
The element type.

Ostr
An object of type `basic_ostream`.

Tr
Character traits.

Return Value

An object of type `basic_ostream`.

Remarks

The manipulator calls *Ostr*.put(*Elem*('\0')). It returns *Ostr*.

Example

C++

```
// ostream_ends.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "a";
   cout << "b" << ends;
   cout << "c" << endl;
}
```

Output

```
ab c
```

flush

Flushes the buffer.

C++

```
template class<Elem, Tr>
basic_ostream<Elem, Tr>& flush(
   basic_ostream<Elem, Tr>& Ostr);
```

Parameters

Elem

The element type.

Ostr

An object of type `basic_ostream`.

Tr

Character traits.

Return Value

An object of type `basic_ostream`.

Remarks

The manipulator calls *Ostr*.flush. It returns *Ostr*.

Example

C++

```
// ostream_flush.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << "testing" << flush;
}
```

Output

```
testing
```

swap

Exchanges the values of two `basic_ostream` objects.

C++

```
template <class Elem, class Tr>
void swap(
   basic_ostream<Elem, Tr>& left,
   basic_ostream<Elem, Tr>& right);
```

Parameters

Elem
The element type.

Tr
Character traits.

left
An lvalue reference to a `basic_ostream` object.

right
An lvalue reference to a `basic_ostream` object.

Remarks

The template function `swap` executes `left.swap(right)`.

See also

<ostream>

`<ostream>` operators

Article • 12/07/2021

operator<<

`operator<<`

Writes various types to the stream.

C++

```
template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    const Elem* str);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    Elem _Ch);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    const char* str);

template <class _Elem, class _Tr>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>& _Ostr,
    char _Ch);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _ostr,
    char _Ch);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const signed char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    signed char _Ch);
```

```
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const unsigned char* str);

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    unsigned char _Ch);

template <class _Elem, class _Tr, class T>
basic_ostream <_Elem, _Tr>& operator<<(
    basic_ostream<_Elem, _Tr>&& _Ostr,
    Ty val);
```

Parameters

_Ch
A character.

_Elem
The element type.

_Ostr
A `basic_ostream` object.

str
A character string.

_Tr
Character traits.

val
The type

Return Value

The stream.

Remarks

The `basic_ostream` class also defines several insertion operators. For more information, see basic_ostream::operator<<.

The template function

C++

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _ostr,
    const Elem *str);
```

determines the length N = `traits_type::`length(`str`) of the sequence beginning at *str*, and inserts the sequence. If N < `_Ostr.`width, then the function also inserts a repetition of `_Ostr.width` - N fill characters. The repetition precedes the sequence if (`_Ostr`. flags & `adjustfield` != left. Otherwise, the repetition follows the sequence. The function returns *_Ostr*.

The template function

C++

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    Elem _Ch);
```

inserts the element `_Ch`. If 1 < `_Ostr.width`, then the function also inserts a repetition of `_Ostr.width` - 1 fill characters. The repetition precedes the sequence if `_Ostr.flags & adjustfield != left`. Otherwise, the repetition follows the sequence. It returns *_Ostr*.

The template function

C++

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    const char *str);
```

behaves the same as

C++

```
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    const Elem *str);
```

except that each element *_Ch* of the sequence beginning at *str* is converted to an object of type `Elem` by calling `_Ostr.put(_Ostr.widen(_Ch))`.

The template function

```
C++

template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    char _Ch);
```

behaves the same as

```
C++

template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    Elem _Ch);
```

except that *_Ch* is converted to an object of type `Elem` by calling `_Ostr.put( _Ostr.widen( _Ch ))`.

The template function

```
C++

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const char *str);
```

behaves the same as

```
C++

template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    const Elem *str);
```

(It does not have to widen the elements before inserting them.)

The template function

```
```

```C++
template <class _Tr>
basic_ostream<char, Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    char _Ch);
```

behaves the same as

```C++
template <class _Elem, class _Tr>
basic_ostream<Elem, _Tr>& operator<<(
    basic_ostream<Elem, _Tr>& _Ostr,
    Elem _Ch);
```

(It does not have to widen *_Ch* before inserting it.)

The template function

```C++
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    const signed char *str);
```

returns `_Ostr << (const char *)str`.

The template function

```C++
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    signed char _Ch);
```

returns `_Ostr << (char)_Ch`.

The template function:

```C++
template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
```

```
    basic_ostream<char, _Tr>& _Ostr,
    const unsigned char *str);
```

returns `_Ostr << (const char *)str`.

The template function:

```
C++

template <class _Tr>
basic_ostream<char, _Tr>& operator<<(
    basic_ostream<char, _Tr>& _Ostr,
    unsigned char _Ch);
```

returns `_Ostr << (char)_Ch`.

The template function:

```
C++

template <class _Elem, class _Tr, class T>
basic_ostream<_Elem, _Tr>& operator<<(
    basic_ostream<char, _Tr>&& _Ostr,
    T val);
```

returns `_Ostr << val` (and converts a RValue Reference to `_Ostr` to an lvalue in the process).

Example

See flush for an example using `operator<<`.

See also

<ostream>

`<ostream>` typedefs

Article • 12/07/2021

ostream
wostream

ostream

Creates a type from basic_ostream that is specialized on **`char`** and `char_traits` specialized on **`char`**.

```
C++
```

```
typedef basic_ostream<char, char_traits<char>> ostream;
```

Remarks

The type is a synonym for class template basic_ostream, specialized for elements of type **`char`** with default character traits.

wostream

Creates a type from basic_ostream that is specialized on **`wchar_t`** and `char_traits` specialized on **`wchar_t`**.

```
C++
```

```
typedef basic_ostream<wchar_t, char_traits<wchar_t>> wostream;
```

Remarks

The type is a synonym for class template basic_ostream, specialized for elements of type **`wchar_t`** with default character traits.

See also

<ostream>

<queue>

Article • 12/07/2021

Defines the class templates priority_queue and queue and several supporting templates.

Requirements

Header: <queue>

Namespace: std

> ⓘ Note
>
> The <queue> library also uses the `#include <initializer_list>` statement.

Members

Operators

Name	Description
operator!=	Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.
operator<	Tests if the queue object on the left side of the operator is less than the queue object on the right side.
operator<=	Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.
operator==	Tests if the queue object on the left side of the operator is equal to the queue object on the right side.
operator>	Tests if the queue object on the left side of the operator is greater than the queue object on the right side.
operator>=	Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

Classes

Name	Description
queue Class	A template container adaptor class that provides a restriction of functionality limiting access to the front and back elements of some underlying container type.
priority_queue Class	A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`<queue>` operators

Article • 12/07/2021

operator!=

Tests if the queue object on the left side of the operator is not equal to the queue object on the right side.

```
C++
```

```
bool operator!=(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queues are not equal; **`false`** if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// queue_op_ne.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   // Declares queues with list base containers
   queue <int, list<int> > q1, q2, q3;

   // The following line would have caused an error because vector
   // does not support pop_front( ) and so cannot be adapted
   // by queue as a base container
   // queue <int, vector<int> > q1, q2, q3;

   q1.push( 1 );
   q2.push( 1 );
   q2.push( 2 );
   q3.push( 1 );

   if ( q1 != q2 )
      cout << "The queues q1 and q2 are not equal." << endl;
   else
      cout << "The queues q1 and q2 are equal." << endl;

   if ( q1 != q3 )
      cout << "The queues q1 and q3 are not equal." << endl;
   else
      cout << "The queues q1 and q3 are equal." << endl;
}
```

Output

```
The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.
```

`operator<`

Tests if the queue object on the left side of the operator is less than the queue object on the right side.

C++

```
bool operator<(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queue on the left side of the operator is less than and not equal to the queue on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// queue_op_lt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queues with default deque base container
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );

   if ( q1 < q2 )
      cout << "The queue q1 is less than the queue q2." << endl;
   else
      cout << "The queue q1 is not less than the queue q2." << endl;

   if ( q1 < q3 )
      cout << "The queue q1 is less than the queue q3." << endl;
```

```
   else
      cout << "The queue q1 is not less than the queue q3." << endl;
}
```

Output

```
The queue q1 is less than the queue q2.
The queue q1 is not less than the queue q3.
```

`operator<=`

Tests if the queue object on the left side of the operator is less than or equal to the queue object on the right side.

C++

```
bool operator<=(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// queue_op_le.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 5 );
   q1.push( 10 );
   q2.push( 1 );
   q2.push( 2 );
   q3.push( 5 );
   q3.push( 10 );

   if ( q1 <= q2 )
      cout << "The queue q1 is less than or equal to "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is greater than "
           << "the queue q2." << endl;

   if ( q1 <= q3 )
      cout << "The queue q1 is less than or equal to "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is greater than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is greater than the queue q2.
The queue q1 is less than or equal to the queue q3.
```

operator==

Tests if the queue object on the left side of the operator is equal to queue object on the right side.

C++

```
bool operator==(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queues are not equal; `false` if queues are equal.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// queue_op_eq.cpp
// compile with: /EHsc
#include <queue>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queues with list base containers
   queue <int, list<int> > q1, q2, q3;

   // The following line would have caused an error because vector
   // does not support pop_front( ) and so cannot be adapted
   // by queue as a base container
   // queue <int, vector<int> > q1, q2, q3;

   q1.push( 1 );
   q2.push( 2 );
   q3.push( 1 );

   if ( q1 != q2 )
      cout << "The queues q1 and q2 are not equal." << endl;
   else
```

```
      cout << "The queues q1 and q2 are equal." << endl;

   if ( q1 != q3 )
      cout << "The queues q1 and q3 are not equal." << endl;
   else
      cout << "The queues q1 and q3 are equal." << endl;
}
```

Output

```
The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.
```

`operator>`

Tests if the queue object on the left side of the operator is greater than the queue object on the right side.

C++

```
bool operator>(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. The greater-than relationship between two queue objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// queue_op_gt.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q1.push( 3 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );

   if ( q1 > q2 )
      cout << "The queue q1 is greater than "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is not greater than "
           << "the queue q2." << endl;

   if ( q1> q3 )
      cout << "The queue q1 is greater than "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is not greater than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is not greater than the queue q2.
The queue q1 is greater than the queue q3.
```

`operator>=`

Tests if the queue object on the left side of the operator is greater than or equal to the queue object on the right side.

C++

```
bool operator>=(const queue <Type, Container>& left, const queue <Type,
Container>& right,);
```

Parameters

left
An object of type `queue`.

right
An object of type `queue`.

Return Value

`true` if the queue on the left side of the operator is strictly less than the queue on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between queue objects is based on a pairwise comparison of their elements. Two queues are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// queue_op_ge.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2, q3;

   q1.push( 1 );
   q1.push( 2 );
   q2.push( 5 );
   q2.push( 10 );
   q3.push( 1 );
   q3.push( 2 );

   if ( q1 >= q2 )
```

```
      cout << "The queue q1 is greater than or equal to "
           << "the queue q2." << endl;
   else
      cout << "The queue q1 is less than "
           << "the queue q2." << endl;

   if ( q1>= q3 )
      cout << "The queue q1 is greater than or equal to "
           << "the queue q3." << endl;
   else
      cout << "The queue q1 is less than "
           << "the queue q3." << endl;
}
```

Output

```
The queue q1 is less than the queue q2.
The queue q1 is greater than or equal to the queue q3.
```

`priority_queue` Class

Article • 02/21/2022

A template container adaptor class that provides a restriction of functionality limiting access to the top element of some underlying container type, which is always the largest or of the highest priority. New elements can be added to the `priority_queue` and the top element of the `priority_queue` can be inspected or removed.

Syntax

C++

```
template <class Type, class Container= vector <Type>, class Compare= less
<typename Container ::value_type>>
class priority_queue
```

Parameters

`Type`

The element data type to be stored in the `priority_queue`.

`Container`

The type of the underlying container used to implement the `priority_queue`.

`Compare`

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the `priority_queue`. This argument is optional and the binary predicate `less<typename Container::value_type>` is the default value.

Remarks

The elements of class `Type` stipulated in the first template parameter of a queue object are synonymous with value_type and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it's possible to copy objects of that type and to assign values to variables of that type.

The `priority_queue` orders the sequence it controls by calling a stored function object of class `Traits`. In general, the elements need be merely less than comparable to

establish this order: so that, given any two elements, it may be determined either that they're equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense.

Suitable underlying container classes for `priority_queue` include deque Class and the default vector Class or any other sequence container that supports the operations of `front`, `push_back`, and `pop_back` and a random-access iterator. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

Adding elements to and removing elements from a `priority_queue` both have logarithmic complexity. Accessing elements in a `priority_queue` has constant complexity.

There are three types of container adaptors defined by the C++ Standard Library: `stack`, `queue`, and `priority_queue`. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The stack Class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the `stack` class.

- The queue Class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason for using the `queue` class.

- The `priority_queue` class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they're arranged by age, height, or some other criterion.

Constructors

Constructor	Description

Constructor	Description
priority_queue	Constructs a `priority_queue` that is empty or that is a copy of a range of a base container object or of other `priority_queue`.

Typedefs

Type name	Description
container_type	A type that provides the base container to be adapted by a `priority_queue`.
size_type	An unsigned integer type that can represent the number of elements in a `priority_queue`.
value_type	A type that represents the type of object stored as an element in a `priority_queue`.

Member functions

Member function	Description
empty	Tests if the `priority_queue` is empty.
pop	Removes the largest element of the `priority_queue` from the top position.
push	Adds an element to the priority queue based on the priority of the element from `operator<`.
size	Returns the number of elements in the `priority_queue`.
top	Returns a const reference to the largest element at the top of the `priority_queue`.

Requirements

Header: `<queue>`

Namespace: `std`

`priority_queue::container_type`

A type that provides the base container to be adapted.

```
C++
```

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. The C++ Standard Library sequence container class `deque` and the default class `vector` meet the requirements to be used as the base container for a `priority_queue` object. User-defined types satisfying the requirements may also be used.

For more information on `Container`, see the Remarks section of the priority_queue Class topic.

Example

See the example for priority_queue for an example of how to declare and use `container_type`.

`priority_queue::empty`

Tests if a `priority_queue` is empty.

```
C++
```

```
bool empty() const;
```

Return Value

`true` if the `priority_queue` is empty; **`false`** if the `priority_queue` is nonempty.

Example

```
C++
```

```
// pqueue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
```

```
{
   using namespace std;

   // Declares priority_queues with default deque base container
   priority_queue <int> q1, s2;

   q1.push( 1 );

   if ( q1.empty( ) )
      cout << "The priority_queue q1 is empty." << endl;
   else
      cout << "The priority_queue q1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The priority_queue s2 is empty." << endl;
   else
      cout << "The priority_queue s2 is not empty." << endl;
}
```

Output

```
The priority_queue q1 is not empty.
The priority_queue s2 is empty.
```

`priority_queue::pop`

Removes the largest element of the `priority_queue` from the top position.

C++

```
void pop();
```

Remarks

The `priority_queue` must be nonempty to apply the member function. The top of the `priority_queue` is always occupied by the largest element in the container.

Example

C++

```
// pqueue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   priority_queue <int> q1, s2;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   priority_queue <int>::size_type i, iii;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   const int& ii = q1.top( );
   cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;

   q1.pop( );

   iii = q1.size( );
   cout << "After a pop, the priority_queue length is "
        << iii << "." << endl;

   const int& iv = q1.top( );
   cout << "After a pop, the element at the top of the "
        << "priority_queue is " << iv << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
After a pop, the priority_queue length is 2.
After a pop, the element at the top of the priority_queue is 20.
```

`priority_queue::priority_queue`

Constructs a `priority_queue` that is empty or that is a copy of a range of a base container object or of another `priority_queue`.

C++

```
priority_queue();

explicit priority_queue(const Traits& _comp);

priority_queue(const Traits& _comp, const container_type& _Cont);

priority_queue(const priority_queue& right);
```

```
template <class InputIterator>
priority_queue(InputIterator first, InputIterator last);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Traits&
_comp);

template <class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Traits& _comp,
const container_type& _Cont);
```

Parameters

`_comp`
The comparison function of type **`constTraits`** used to order the elements in the `priority_queue`, which defaults to compare function of the base container.

`_Cont`
The base container of which the constructed `priority_queue` is to be a copy.

`right`
The `priority_queue` of which the constructed set is to be a copy.

`first`
The position of the first element in the range of elements to be copied.

`last`
The position of the first element beyond the range of elements to be copied.

Remarks

Each of the first three constructors specifies an empty initial `priority_queue`, the second also specifying the type of comparison function (`comp`) to be used in establishing the order of the elements and the third explicitly specifying the `container_type` (`_Cont`) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the `priority_queue right`.

The last three constructors copy the range `[first, last)` of some container and use the values to initialize a `priority_queue` with increasing explicitness in specifying the type of comparison function of class `Traits` and `container_type`.

Example

C++

```
// pqueue_ctor.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <deque>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function declares priority_queue
   // with a default vector base container
   priority_queue <int> q1;
   cout << "q1 = ( ";
   while ( !q1.empty( ) )
   {
      cout << q1.top( ) << " ";
      q1.pop( );
   }
   cout << ")" << endl;

   // Explicitly declares a priority_queue with nondefault
   // deque base container
   priority_queue <int, deque <int> > q2;
   q2.push( 5 );
   q2.push( 15 );
   q2.push( 10 );
   cout << "q2 = ( ";
   while ( !q2.empty( ) )
   {
      cout << q2.top( ) << " ";
      q2.pop( );
   }
   cout << ")" << endl;

   // This method of printing out the elements of a priority_queue
   // removes the elements from the priority queue, leaving it empty
   cout << "After printing, q2 has " << q2.size( ) << " elements." << endl;

   // The third member function declares a priority_queue
   // with a vector base container and specifies that the comparison
   // function greater is to be used for ordering elements
   priority_queue <int, vector<int>, greater<int> > q3;
   q3.push( 2 );
   q3.push( 1 );
   q3.push( 3 );
   cout << "q3 = ( ";
   while ( !q3.empty( ) )
```

```
{
   cout << q3.top( ) << " ";
   q3.pop( );
}
cout << ")" << endl;

// The fourth member function declares a priority_queue and
// initializes it with elements copied from another container:
// first, inserting elements into q1, then copying q1 elements into q4
q1.push( 100 );
q1.push( 200 );
priority_queue <int> q4( q1 );
cout << "q4 = ( ";
while ( !q4.empty( ) )
{
   cout << q4.top( ) << " ";
   q4.pop( );
}
cout << ")" << endl;

// Creates an auxiliary vector object v5 to be used to initialize q5
vector <int> v5;
vector <int>::iterator v5_Iter;
v5.push_back( 10 );
v5.push_back( 30 );
v5.push_back( 20 );
cout << "v5 = ( " ;
for ( v5_Iter = v5.begin( ) ; v5_Iter != v5.end( ) ; v5_Iter++ )
   cout << *v5_Iter << " ";
cout << ")" << endl;

// The fifth member function declares and
// initializes a priority_queue q5 by copying the
// range v5[ first,  last) from vector v5
priority_queue <int> q5( v5.begin( ), v5.begin( ) + 2 );
cout << "q5 = ( ";
while ( !q5.empty( ) )
{
   cout << q5.top( ) << " ";
   q5.pop( );
}
cout << ")" << endl;

// The sixth member function declares a priority_queue q6
// with a comparison function greater and initializes q6
// by copying the range v5[ first,  last) from vector v5
priority_queue <int, vector<int>, greater<int> >
   q6( v5.begin( ), v5.begin( ) + 2 );
cout << "q6 = ( ";
while ( !q6.empty( ) )
{
   cout << q6.top( ) << " ";
   q6.pop( );
}
```

```
    cout << ")" << endl;
}
```

`priority_queue::push`

Adds an element to the priority queue based on the priority of the element from `operator<`.

C++

```
void push(const Type& val);
```

Parameters

`val`
The element added to the top of the `priority_queue`.

Remarks

The top of the `priority_queue` is the position occupied by the largest element in the container.

Example

C++

```
// pqueue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue<int> q1;

   q1.push( 10 );
   q1.push( 30 );
   q1.push( 20 );

   priority_queue<int>::size_type i;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;
```

```
    const int& ii = q1.top( );
    cout << "The element at the top of the priority_queue is "
         << ii << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

`priority_queue::size`

Returns the number of elements in the `priority_queue`.

C++

```
size_type size() const;
```

Return Value

The current length of the `priority_queue`.

Example

C++

```
// pqueue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue <int> q1, q2;
   priority_queue <int>::size_type i;

   q1.push( 1 );
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   q1.push( 2 );
   i = q1.size( );
   cout << "The priority_queue length is now " << i << "." << endl;
}
```

Output

```
The priority_queue length is 1.
The priority_queue length is now 2.
```

`priority_queue::size_type`

An unsigned integer type that can represent the number of elements in a `priority_queue`.

C++

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the `size_type` of the base container adapted by the `priority_queue`.

Example

See the example for size for an example of how to declare and use `size_type`.

`priority_queue::top`

Returns a const reference to the largest element at the top of the `priority_queue`.

C++

```
const_reference top() const;
```

Return Value

A reference to the largest element, as determined by the `Traits` function, object of the `priority_queue`.

Remarks

The `priority_queue` must be nonempty to apply the member function.

Example

C++

```
// pqueue_top.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   priority_queue<int> q1;

   q1.push( 10 );
   q1.push( 30 );
   q1.push( 20 );

   priority_queue<int>::size_type i;
   i = q1.size( );
   cout << "The priority_queue length is " << i << "." << endl;

   const int& ii = q1.top( );
   cout << "The element at the top of the priority_queue is "
        << ii << "." << endl;
}
```

Output

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

`priority_queue::value_type`

A type that represents the type of object stored as an element in a `priority_queue`.

C++

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the `value_type` of the base container adapted by the `priority_queue`.

Example

C++

```
// pqueue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares priority_queues with default deque base container
   priority_queue<int>::value_type AnInt;

   AnInt = 69;
   cout << "The value_type is AnInt = " << AnInt << endl;

   priority_queue<int> q1;
   q1.push( AnInt );
   cout << "The element at the top of the priority_queue is "
        << q1.top( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the top of the priority_queue is 69.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`queue` Class

Article • 01/25/2022

A template container adaptor class that provides a restriction of functionality for some underlying container type, limiting access to the front and back elements. Elements can be added at the back or removed from the front, and elements can be inspected at either end of the `queue`.

Syntax

```
C++
```

```
template <class Type, class Container = deque <Type>>
class queue
```

Parameters

`Type`

The element data type to be stored in the `queue`.

`Container`

The type of the underlying container used to implement the `queue`.

Remarks

The elements of class `Type` stipulated in the first template parameter of a `queue` object are synonymous with value_type and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for `queue` include deque and list, or any other sequence container that supports the operations of `front`, `back`, `push_back`, and `pop_front`. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The `queue` objects are equality comparable if and only if the elements of class `Type` are equality comparable, and are less-than comparable if and only if the elements of class

`Type` are less-than comparable.

There are three types of container adaptors defined by the C++ Standard Library: `stack`, `queue`, and `priority_queue`. Each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.

- The stack class supports a last-in, first-out (LIFO) data structure. A good analog to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the `stack` class.
- The `queue` class supports a first-in, first-out (FIFO) data structure. A good analog to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the `front` and `back` elements in this way is the reason for using the `queue` class.
- The priority_queue class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analog to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Members

Constructors

Name	Description
queue	Constructs a `queue` that is empty or that is a copy of a base container object.

Typedefs

Name	Description
container_type	A type that provides the base container to be adapted by the `queue`.
size_type	An unsigned integer type that can represent the number of elements in a `queue`.
value_type	A type that represents the type of object stored as an element in a `queue`.

Functions

Name	Description
back	Returns a reference to the last and most recently added element at the back of the `queue`.
empty	Tests if the `queue` is empty.
front	Returns a reference to the first element at the front of the `queue`.
pop	Removes an element from the front of the `queue`.
push	Adds an element to the back of the `queue`.
size	Returns the number of elements in the `queue`.

back

Returns a reference to the last and most recently added element at the back of the `queue`.

```
C++
```

```
reference back();

const_reference back() const;
```

Return Value

The last element of the `queue`. If the `queue` is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the `queue` object cannot be modified. If the return value of `back` is assigned to a `reference`, the `queue` object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty `queue`. See Checked Iterators for more information.

Example

C++

```
// queue_back.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 11 );

   int& i = q1.back( );
   const int& ii = q1.front( );

   cout << "The integer at the back of queue q1 is " << i
        << "." << endl;
   cout << "The integer at the front of queue q1 is " << ii
        << "." << endl;
}
```

`container_type`

A type that provides the base container to be adapted.

C++

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. Two C++ Standard Library sequence container classes—the `list` class and the default `deque` class—meet the requirements to be used as the base container for a `queue` object. User-defined types satisfying the requirements may also be used.

For more information on `Container`, see the Remarks section of the queue Class topic.

Example

See the example for queue for an example of how to declare and use `container_type`.

`empty`

Tests if a `queue` is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the `queue` is empty; **`false`** if the `queue` is nonempty.

Example

C++

```
// queue_empty.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
using namespace std;

   // Declares queues with default deque base container
   queue <int> q1, q2;

   q1.push( 1 );

   if ( q1.empty( ) )
      cout << "The queue q1 is empty." << endl;
   else
      cout << "The queue q1 is not empty." << endl;

   if ( q2.empty( ) )
      cout << "The queue q2 is empty." << endl;
   else
      cout << "The queue q2 is not empty." << endl;
}
```

Output

```
The queue q1 is not empty.
The queue q2 is empty.
```

`front`

Returns a reference to the first element at the front of the `queue`.

```
C++
```

```
reference front();

const_reference front() const;
```

Return Value

The first element of the `queue`. If the `queue` is empty, the return value is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the `queue` object cannot be modified. If the return value of `front` is assigned to a `reference`, the `queue` object can be modified.

The member function returns a `reference` to the first element of the controlled sequence, which must be nonempty.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty `queue`. See Checked Iterators for more information.

Example

```
C++
```

```
// queue_front.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main() {
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
```

```
    i = q1.size( );
    cout << "The queue length is " << i << "." << endl;

    int& ii = q1.back( );
    int& iii = q1.front( );

    cout << "The integer at the back of queue q1 is " << ii
         << "." << endl;
    cout << "The integer at the front of queue q1 is " << iii
         << "." << endl;
}
```

pop

Removes an element from the front of the `queue`.

C++

```
void pop();
```

Remarks

The `queue` must be nonempty to apply the member function. The top of the `queue` is the position occupied by the most recently added element and is the last element at the end of the container.

Example

C++

```
// queue_pop.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, s2;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
   i = q1.size( );
```

```
    cout << "The queue length is " << i << "." << endl;

    i = q1.front( );
    cout << "The element at the front of the queue is "
         << i << "." << endl;

    q1.pop( );

    i = q1.size( );
    cout << "After a pop the queue length is "
         << i << "." << endl;

    i = q1. front ( );
    cout << "After a pop, the element at the front of the queue is "
         << i << "." << endl;
}
```

Output

```
The queue length is 3.
The element at the front of the queue is 10.
After a pop the queue length is 2.
After a pop, the element at the front of the queue is 20.
```

push

Adds an element to the back of the `queue`.

C++

```
void push(const Type& val);
```

Parameters

`val`

The element added to the back of the `queue`.

Remarks

The back of the `queue` is the position occupied by the most recently added element and is the last element at the end of the container.

Example

C++

```
// queue_push.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1;

   q1.push( 10 );
   q1.push( 20 );
   q1.push( 30 );

   queue <int>::size_type i;
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   i = q1.front( );
   cout << "The element at the front of the queue is "
        << i << "." << endl;
}
```

Output

```
The queue length is 3.
The element at the front of the queue is 10.
```

queue

Constructs a `queue` that is empty or that is a copy of a base container object.

C++

```
queue();

explicit queue(const container_type& right);
```

Parameters

`right`

The **`const`** container of which the constructed `queue` is to be a copy.

Remarks

The default base container for `queue` is `deque`. You can also specify `list` as a base container, but you cannot specify `vector`, because it lacks the required `pop_front` member function.

Example

C++

```
// queue_queue.cpp
// compile with: /EHsc
#include <queue>
#include <vector>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares queue with default deque base container
   queue <char> q1;

   // Explicitly declares a queue with deque base container
   queue <char, deque<char> > q2;

   // These lines don't cause an error, even though they
   // declares a queue with a vector base container
   queue <int, vector<int> > q3;
   q3.push( 10 );
   // but the following would cause an error because vector has
   // no pop_front member function
   // q3.pop( );

   // Declares a queue with list base container
   queue <int, list<int> > q4;

   // The second member function copies elements from a container
   list<int> li1;
   li1.push_back( 1 );
   li1.push_back( 2 );
   queue <int, list<int> > q5( li1 );
   cout << "The element at the front of queue q5 is "
        << q5.front( ) << "." << endl;
   cout << "The element at the back of queue q5 is "
        << q5.back( ) << "." << endl;
}
```

Output

```
The element at the front of queue q5 is 1.
The element at the back of queue q5 is 2.
```

`size`

Returns the number of elements in the `queue`.

C++

```
size_type size() const;
```

Return Value

The current length of the `queue`.

Example

C++

```
// queue_size.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
   using namespace std;
   queue <int> q1, q2;
   queue <int>::size_type i;

   q1.push( 1 );
   i = q1.size( );
   cout << "The queue length is " << i << "." << endl;

   q1.push( 2 );
   i = q1.size( );
   cout << "The queue length is now " << i << "." << endl;
}
```

Output

```
The queue length is 1.
The queue length is now 2.
```

`size_type`

An unsigned integer type that can represent the number of elements in a `queue`.

```
C++
```

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for the `size_type` of the base container adapted by the `queue`.

Example

See the example for queue::front for an example of how to declare and use `size_type`.

`value_type`

A type that represents the type of object stored as an element in a `queue`.

```
C++
```

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for the `value_type` of the base container adapted by the `queue`.

Example

```
C++
```

```
// queue_value_type.cpp
// compile with: /EHsc
#include <queue>
#include <iostream>

int main( )
{
using namespace std;

   // Declares queues with default deque base container
```

```
    queue<int>::value_type AnInt;

    AnInt = 69;
    cout << "The value_type is AnInt = " << AnInt << endl;

    queue<int> q1;
    q1.push(AnInt);
    cout << "The element at the front of the queue is "
         << q1.front( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the front of the queue is 69.
```

See also

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<random>`

Article • 06/06/2022

Defines facilities for random number generation, allowing creation of uniformly distributed random numbers.

Requirements

Header: `<random>`

Namespace: std

> ⓘ **Note**
>
> The `<random>` library uses the `#include <initializer_list>' statement.

Summary

A *random number generator* is an object that produces a sequence of pseudo-random values. A generator that produces values that are uniformly distributed in a specified range is a *Uniform Random Number Generator* (URNG). A class template designed to function as a URNG is referred to as an *engine* if that class has certain common traits, which are discussed later in this article. A URNG can be—and usually is—combined with a *distribution* by passing the URNG as an argument to the distribution's `operator()` to produce values that are distributed in a manner that is defined by the distribution.

These links jump to the major sections of this article:

- Examples
- Categorized Listing
- Engines and Distributions
- Remarks

Quick Tips

Here are some tips to keep in mind when using `<random>`:

- For most purposes, URNGs produce raw bits that must be shaped by distributions. (A notable exception to this is std::shuffle() because it uses a URNG directly.)
- A single instantiation of a URNG or distribution cannot safely be called concurrently because running a URNG or distribution is a modifying operation. For more information, see Thread Safety in the C++ Standard Library.
- Predefined typedefs of several engines are provided; this is the preferred way to create a URNG if an engine is being used.
- The most useful pairing for most applications is the `mt19937` engine with `uniform_int_distribution`, as shown in the code example later in this article.

There are many options to choose from in the `<random>` header, and any of them is preferable to the outdated C Runtime function `rand()`. For information about what's wrong with `rand()` and how `<random>` addresses these shortcomings, see this video.

Examples

The following code example shows how to generate some random numbers in this case five of them using a generator created with non-deterministic seed.

C++

```
#include <random>
#include <iostream>

using namespace std;

int main()
{
    random_device rd;   // non-deterministic generator
    mt19937 gen(rd());  // to seed mersenne twister.
                        // replace the call to rd() with a
                        // constant value to get repeatable
                        // results.

    for (int i = 0; i < 5; ++i) {
        cout << gen() << " "; // print the raw output of the generator.
    }
    cout << endl;
}
```

Output

```
2430338871 3531691818 2723770500 3252414483 3632920437
```

While these are high quality random numbers and different every time this program is run, they are not necessarily in a useful range. To control the range, use a uniform distribution as shown in the following code:

C++

```
#include <random>
#include <iostream>

using namespace std;

int main()
{
    random_device rd;   // non-deterministic generator
    mt19937 gen(rd());  // to seed mersenne twister.
    uniform_int_distribution<> dist(1,6); // distribute results between 1
and 6 inclusive.

    for (int i = 0; i < 5; ++i) {
        cout << dist(gen) << " "; // pass the generator to the distribution.
    }
    cout << endl;
}
```

Output

```
5 1 6 1 2
```

The next code example shows a more realistic set of use cases with uniformly distributed random number generators shuffling the contents of a vector and an array.

C++

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <array>
#include <iostream>
#include <random>
#include <string>
#include <vector>
#include <functional> // ref()

using namespace std;

template <typename C> void print(const C& c) {
    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
```

```
}

template <class URNG>
void test(URNG& urng) {

    // Uniform distribution used with a vector
    // Distribution is [-5, 5] inclusive
    uniform_int_distribution<int> dist(-5, 5);
    vector<int> v;

    for (int i = 0; i < 20; ++i) {
        v.push_back(dist(urng));
    }

    cout << "Randomized vector: ";
    print(v);

    // Shuffle an array
    // (Notice that shuffle() takes a URNG, not a distribution)
    array<string, 26> arr = { { "H", "He", "Li", "Be", "B", "C", "N", "O",
"F",
        "Ne", "Na", "Mg", "Al", "Si", "P", "S", "Cl", "Ar", "K", "Ca", "Sc",
        "Ti", "V", "Cr", "Mn", "Fe" } };

    shuffle(arr.begin(), arr.end(), urng);

    cout << "Randomized array: ";
    print(arr);
    cout << "--" << endl;
}

int main()
{
    // First run: non-seedable, non-deterministic URNG random_device
    // Slower but crypto-secure and non-repeatable.
    random_device rd;
    cout << "Using random_device URNG:" << endl;
    test(rd);

    // Second run: simple integer seed, repeatable results
    cout << "Using constant-seed mersenne twister URNG:" << endl;
    mt19937 engine1(12345);
    test(engine1);

    // Third run: random_device as a seed, different each run
    // (Desirable for most purposes)
    cout << "Using non-deterministic-seed mersenne twister URNG:" << endl;
    mt19937 engine2(rd());
    test(engine2);

    // Fourth run: "warm-up" sequence as a seed, different each run
    // (Advanced uses, allows more than 32 bits of randomness)
    cout << "Using non-deterministic-seed \"warm-up\" sequence mersenne
twister URNG:" << endl;
    array<unsigned int, mt19937::state_size> seed_data;
```

```
    generate_n(seed_data.begin(), seed_data.size(), ref(rd));
    seed_seq seq(begin(seed_data), end(seed_data));
    mt19937 engine3(seq);
    test(engine3);
}
```

Output

```
Using random_device URNG:
Randomized vector: 5 -4 2 3 0 5 -2 0 4 2 -1 2 -4 -3 1 4 4 1 2 -2
Randomized array: O Li V K C Ti N Mg Ne Sc Cl B Cr Mn Ca Al F P Na Be Si Ar
Fe S He H
--
Using constant-seed mersenne twister URNG:
Randomized vector: 3 -1 -5 0 0 5 3 -4 -3 -4 1 -3 0 -3 -2 -4 5 1 -1 -1
Randomized array: Al O Ne Si Na Be C N Cr Mn H V F Sc Mg Fe K Ca S Ti B P Ar
Cl Li He
--
Using non-deterministic-seed mersenne twister URNG:
Randomized vector: 5 -4 0 2 1 -2 4 4 -4 0 0 4 -5 4 -5 -1 -3 0 0 3
Randomized array: Si Fe Al Ar Na P B Sc H F Mg Li C Ti He N Mn Be O Ca Cr V
K Ne Cl S
--
Using non-deterministic-seed "warm-up" sequence mersenne twister URNG:
Randomized vector: -1 3 -2 4 1 3 0 -5 5 -5 0 0 5 0 -3 3 -4 2 5 0
Randomized array: Si C Sc H Na O S Cr K Li Al Ti Cl B Mn He Fe Ne Be Ar V P
Ca N Mg F
--
```

This code demonstrates two different randomizations—randomize a vector of integers and shuffle an array of indexed data—with a test template function. The first call to the test function uses the crypto-secure, non-deterministic, not-seedable, non-repeatable URNG `random_device`. The second test run uses `mersenne_twister_engine` as URNG, with a deterministic 32-bit constant seed, which means the results are repeatable. The third test run seeds `mersenne_twister_engine` with a 32-bit non-deterministic result from `random_device`. The fourth test run expands on this by using a seed sequence filled with `random_device` results, which effectively gives more than 32-bit non-deterministic randomness (but still not crypto-secure). For more information, read on.

Categorized Listing

Uniform Random Number Generators

URNGs are often described in terms of these properties:

1. **Period length**: How many iterations it takes to repeat the sequence of numbers generated. The longer the better.
2. **Performance**: How quickly numbers can be generated and how much memory it takes. The smaller the better.
3. **Quality**: How close to true random numbers the generated sequence is. This is often called "*randomness*".

The following sections list the uniform random number generators (URNGs) provided in the `<random>` header.

Non-Deterministic Generator

random_device Class
Generates a non-deterministic, cryptographically secure random sequence by using an external device. Usually used to seed an engine. Low performance, very high quality. For more information, see Remarks.

Engine Typedefs with Predefined Parameters

For instantiating engines and engine adaptors. For more information, see Engines and Distributions.

- `default_random_engine` The default engine.

```
C++

typedef mt19937 default_random_engine;
```

- `knuth_b` Knuth engine.

```
C++

typedef shuffle_order_engine<minstd_rand0, 256> knuth_b;
```

- `minstd_rand0` 1988 minimal standard engine (Lewis, Goodman, and Miller, 1969).

```
C++

typedef linear_congruential_engine<unsigned int, 16807, 0, 2147483647>
minstd_rand0;
```

- `minstd_rand` Updated minimal standard engine `minstd_rand0` (Park, Miller, and Stockmeyer, 1993).

```
C++

typedef linear_congruential_engine<unsigned int, 48271, 0, 2147483647>
minstd_rand;
```

- `mt19937` 32-bit Mersenne twister engine (Matsumoto and Nishimura, 1998).

```
C++

typedef mersenne_twister_engine<
    unsigned int, 32, 624, 397,
    31, 0x9908b0df,
    11, 0xffffffff,
    7, 0x9d2c5680,
    15, 0xefc60000,
    18, 1812433253> mt19937;
```

- `mt19937_64` 64-bit Mersenne twister engine (Matsumoto and Nishimura, 2000).

```
C++

typedef mersenne_twister_engine<
    unsigned long long, 64, 312, 156,
    31, 0xb5026f5aa96619e9ULL,
    29, 0x5555555555555555ULL,
    17, 0x71d67fffeda60000ULL,
    37, 0xfff7eee000000000ULL,
    43, 6364136223846793005ULL> mt19937_64;
```

- `ranlux24` 24-bit RANLUX engine (Martin Lüscher and Fred James, 1994).

```
C++

typedef discard_block_engine<ranlux24_base, 223, 23> ranlux24;
```

- `ranlux24_base` Used as a base for `ranlux24`.

```
C++

typedef subtract_with_carry_engine<unsigned int, 24, 10, 24>
ranlux24_base;
```

- `ranlux48` 48-bit RANLUX engine (Martin Lüscher and Fred James, 1994).

```
C++

typedef discard_block_engine<ranlux48_base, 389, 11> ranlux48;
```

- `ranlux48_base` Used as a base for `ranlux48`.

```
C++

typedef subtract_with_carry_engine<unsigned long long, 48, 5, 12>
ranlux48_base;
```

Engine Templates

Engine templates are used as standalone URNGs or as base engines passed to engine adaptors. Usually these are instantiated with a predefined engine typedef and passed to a distribution. For more information, see the Engines and Distributions section.

Name	Description
linear_congruential_engine Class	Generates a random sequence by using the linear congruential algorithm. Most simplistic and lowest quality.
mersenne_twister_engine Class	Generates a random sequence by using the Mersenne twister algorithm. Most complex, and is highest quality except for the random_device class. Very fast performance.
subtract_with_carry_engine Class	Generates a random sequence by using the subtract-with-carry algorithm. An improvement on `linear_congruential_engine`, but much lower quality and performance than `mersenne_twister_engine`.

Engine Adaptor Templates

Engine adaptors are templates that adapt other (base) engines. Usually these are instantiated with a predefined engine typedef and passed to a distribution. For more information, see the Engines and Distributions section.

Name	Description
discard_block_engine Class	Generates a random sequence by discarding values returned by its base engine.
independent_bits_engine Class	Generates a random sequence with a specified number of bits by repacking bits from the values returned by its base engine.

Name	Description
shuffle_order_engine Class	Generates a random sequence by reordering the values returned from its base engine.

[Engine Templates]

Random Number Distributions

The following sections list the distributions provided in the `<random>` header. Distributions are a post-processing mechanism, usually using URNG output as input and distributing the output by a defined statistical probability density function. For more information, see the Engines and Distributions section.

Uniform Distributions

Name	Description
uniform_int_distribution Class	Produces a uniform integer value distribution across a range in the closed interval [a, b] (inclusive-inclusive).
uniform_real_distribution Class	Produces a uniform real (floating-point) value distribution across a range in the half-open interval [a, b) (inclusive-exclusive).
generate_canonical	Produces an even distribution of real (floating point) values of a given precision across [0, 1) (inclusive-exclusive).

[Random Number Distributions]

Bernoulli Distributions

Name	Description
bernoulli_distribution Class	Produces a Bernoulli distribution of `bool` values.
binomial_distribution Class	Produces a binomial distribution of integer values.
geometric_distribution Class	Produces a geometric distribution of integer values.
negative_binomial_distribution Class	Produces a negative binomial distribution of integer values.

[Random Number Distributions]

Normal Distributions

Name	Description
cauchy_distribution Class	Produces a Cauchy distribution of real (floating point) values.
chi_squared_distribution Class	Produces a chi-squared distribution of real (floating point) values.
fisher_f_distribution Class	Produces an F-distribution (also known as Snedecor's F distribution or the Fisher-Snedecor distribution) of real (floating point) values.
lognormal_distribution Class	Produces a log-normal distribution of real (floating point) values.
normal_distribution Class	Produces a normal (Gaussian) distribution of real (floating point) values.
student_t_distribution Class	Produces a Student's *t*-distribution of real (floating point) values.

[Random Number Distributions]

Poisson Distributions

Name	Description
exponential_distribution Class	Produces an exponential distribution of real (floating point) values.
extreme_value_distribution Class	Produces an extreme value distribution of real (floating point) values.
gamma_distribution Class	Produces a gamma distribution of real (floating point) values.
poisson_distribution Class	Produces a Poisson distribution of integer values.
weibull_distribution Class	Produces a Weibull distribution of real (floating point) values.

[Random Number Distributions]

Sampling Distributions

Name	Description
discrete_distribution Class	Produces a discrete integer distribution.

Name	Description
piecewise_constant_distribution Class	Produces a piecewise constant distribution of real (floating point) values.
piecewise_linear_distribution Class	Produces a piecewise linear distribution of real (floating point) values.

[Random Number Distributions]

Utility Functions

This section lists the general utility functions provided in the `<random>` header.

Name	Description
seed_seq Class	Generates a non-biased scrambled seed sequence. Used to avoid replication of random variate streams. Useful when many URNGs are instantiated from engines.

Operators

This section lists the operators provided in the `<random>` header.

Name	Description
`operator==`	Tests whether the URNG on the left side of the operator is equal to the engine on the right side.
`operator!=`	Tests whether the URNG on the left side of the operator is not equal to the engine on the right side.
`operator<<`	Writes state information to a stream.
`operator>>`	Extracts state information from a stream.

Engines and Distributions

Refer to the following sections for information about each of these class template categories defined in `<random>`. Both of these class template categories take a type as an argument and use shared template parameter names to describe the properties of the type that are permitted as an actual argument type, as follows:

- `IntType` indicates a **`short`**, **`int`**, **`long`**, **`long long`**, **`unsigned short`**, **`unsigned int`**, **`unsigned long`**, or **`unsigned long long`**.

- `UIntType` indicates **`unsigned short`**, **`unsigned int`**, **`unsigned long`**, or **`unsigned long long`**.
- `RealType` indicates a **`float`**, **`double`**, or **`long double`**.

Engines

Engine Templates and Engine Adaptor Templates are templates whose parameters customize the generator created.

An *engine* is a class or class template whose instances (generators) act as a source of random numbers uniformly distributed between a minimum and maximum value. An *engine adaptor* delivers a sequence of values that have different randomness properties by taking values produced by some other random number engine and applying an algorithm of some kind to those values.

Every engine and engine adaptor has the following members:

- **`typedef`** `numeric-type result_type` is the type that is returned by the generator's `operator()`. The `numeric-type` is passed as a template parameter on instantiation.
- `result_type operator()` returns values that are uniformly distributed between `min()` and `max()`.
- `result_type min()` returns the minimum value that is returned by the generator's `operator()`. Engine adaptors use the base engine's `min()` result.
- `result_type max()` returns the maximum value that is returned by the generator's `operator()`. When `result_type` is an integral (integer-valued) type, `max()` is the maximum value that can actually be returned (inclusive); when `result_type` is a floating-point (real-valued) type, `max()` is the smallest value greater than all values that can be returned (non-inclusive). Engine adaptors use the base engine's `max()` result.
- `void seed(result_type s)` seeds the generator with seed value `s`. For engines, the signature is `void seed(result_type s = default_seed)` for default parameter support (engine adaptors define a separate `void seed()`, see next subsection).
- `template <class Seq> void seed(Seq& q)` seeds the generator by using a seed_seq`Seq`.
- An explicit constructor with argument `result_type x` that creates a generator seeded as if by calling `seed(x)`.

- An explicit constructor with argument `seed_seq& seq` that creates a generator seeded as if by calling `seed(seq)`.
- `void discard(unsigned long long count)` effectively calls `operator()` `count` times and discards each value.

Engine adaptors additionally support these members (`Engine` is the first template parameter of an engine adaptor, designating the base engine's type):

- A default constructor to initialize the generator as if from the base engine's default constructor.
- An explicit constructor with argument `const Engine& eng`. This is to support copy construction using the base engine.
- An explicit constructor with argument `Engine&& eng`. This is to support move construction using the base engine.
- `void seed()` that initializes the generator with the base engine's default seed value.
- `const Engine& base()` property function that returns the base engine that was used to construct the generator.

Every engine maintains a *state* that determines the sequence of values that will be generated by subsequent calls to `operator()`. The states of two generators instantiated from engines of the same type can be compared by using `operator==` and `operator!=`. If the two states compare as equal, they will generate the same sequence of values. The state of an object can be saved to a stream as a sequence of 32-bit unsigned values by using the `operator<<` of the generator. The state is not changed by saving it. A saved state can be read into generator instantiated from an engine of the same type by using `operator>>`.

Distributions

A Random Number Distributions is a class or class template whose instances transform a stream of uniformly distributed random numbers obtained from an engine into a stream of random numbers that have a particular distribution. Every distribution has the following members:

- **`typedef`** `numeric-type result_type` is the type that is returned by the distribution's `operator()`. The `numeric-type` is passed as a template parameter on instantiation.

- `template <class URNG> result_type operator()(URNG& gen)` returns values that are distributed according to the distribution's definition, by using `gen` as a source of uniformly distributed random values and the stored *parameters of the distribution*.
- `template <class URNG> result_type operator()(URNG& gen, param_type p)` returns values distributed in accordance with the distribution's definition, using `gen` as a source of uniformly distributed random values and the parameters structure `p`.
- **`typedef`** `unspecified-type param_type` is the package of parameters optionally passed to `operator()` and is used in place of the stored parameters to generate its return value.
- A `const param&` constructor initializes the stored parameters from its argument.
- `param_type param() const` gets the stored parameters.
- `void param(const param_type&)` sets the stored parameters from its argument.
- `result_type min()` returns the minimum value that is returned by the distribution's `operator()`.
- `result_type max()` returns the maximum value that is returned by the distribution's `operator()`. When `result_type` is an integral (integer-valued) type, `max()` is the maximum value that can actually be returned (inclusive); when `result_type` is a floating-point (real-valued) type, `max()` is the smallest value greater than all values that can be returned (non-inclusive).
- `void reset()` discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

A parameter structure is an object that stores all of the parameters needed for a distribution. It contains:

- **`typedef`** `distribution-type distribution_type`, which is the type of its distribution.
- One or more constructors that take the same parameter lists as the distribution constructors take.
- The same parameter-access functions as the distribution.
- Equality and inequality comparison operators.

For more information, see the reference subtopics below this one, linked previously in this article.

Remarks

There are two highly useful URNGs in Visual Studio—`mt19937` and `random_device`—as shown in this comparison table:

URNG	Fast	Crypto-secure	Seedable	Deterministic
`mt19937`	Yes	No	Yes	Yes*
`random_device`	No	Yes	No	No

* When provided with a known seed.

Although the ISO C++ Standard does not require `random_device` to be cryptographically secure, in Visual Studio it is implemented to be cryptographically secure. (The term "cryptographically secure" does not imply guarantees, but refers to a minimum level of entropy—and therefore, the level of predictability—a given randomization algorithm provides. For more information, see the Wikipedia article Cryptographically secure pseudorandom number generator.) Because the ISO C++ Standard does not require this, other platforms may implement `random_device` as a simple pseudo-random number generator (not cryptographically secure) and may only be suitable as a seed source for another generator. Check the documentation for those platforms when using `random_device` in cross-platform code.

By definition, `random_device` results are not reproducible, and a side-effect is that it may run significantly slower than other URNGs. Most applications that are not required to be cryptographically secure use `mt19937` or a similar engine, although you may want to seed it with a call to `random_device`, as shown in the code example.

See also

Header Files Reference

`<random>` functions

Article • 12/07/2021

generate_canonical

Returns a floating-point value from a random sequence.

```
C++
```

```
template <class RealType, size_t Bits, class Generator>
RealType generate_canonical(Generator& Gen);
```

Parameters

RealType
The floating point integral type. For possible types, see <random>.

Bits
The number of bits of randomness to use.

Generator
A random number generator class.

Gen
A reference to an instance of a random number generator of type *Generator*.

Remarks

The template function calls `operator()` of *Gen* repeatedly and packs the returned values into a floating-point value `x` of type *RealType* until it has gathered the specified number of mantissa bits in `x`. The specified number is the smaller of *Bits* (which must be nonzero) and the full number of mantissa bits in *RealType*. The first call supplies the lowest-order bits. The function returns `x`.

bernoulli_distribution Class

Article • 08/03/2021

Generates a Bernoulli distribution.

Syntax

C++

```
class bernoulli_distribution
   {
public:
   // types
   typedef bool result_type;
   struct param_type;

   // constructors and reset functions
   explicit bernoulli_distribution(double p = 0.5);
   explicit bernoulli_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   double p() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class describes a distribution that produces values of type `bool`, distributed according to the Bernoulli distribution discrete probability function. The following table links to articles about individual members.

bernoulli_distribution
param_type

The property member `p()` returns the currently stored distribution parameter value `p`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the Bernoulli distribution discrete probability function, see the Wolfram MathWorld article Bernoulli Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic seed
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::bernoulli_distribution distr(p);

    std::cout << "p == " << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<bool, int> histogram;
    for (int i = 0; i < s; ++i) {
```

```
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::boolalpha << std::setw(5) << elem.first << ' ' <<
std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 0.5;
    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a double value for p distribution (where 0.0 <= p <=
1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .45
Enter an integer value for a sample count: 100
p == 0.45
Histogram for 100 samples:
false ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
true ::::::::::::::::::::::::::::::::::::::
```

Requirements

Header: <random>

Namespace: std

bernoulli_distribution::bernoulli_distribution

Constructs the distribution.

C++

```
explicit bernoulli_distribution(double p = 0.5);
explicit bernoulli_distribution(const param_type& parm);
```

Parameters

p
The stored `p` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 ≤ p ≤ 1.0`

The first constructor constructs an object whose stored `p` value holds the value *p*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

bernoulli_distribution::param_type

Contains the parameters of the distribution.

struct param_type { typedef bernoulli_distribution distribution_type; param_type(double p = 0.5); double p() const;

bool operator==(const param_type& right) const; bool operator!=(const param_type& right) const; };

Parameters

p
The stored `p` distribution parameter.

Remarks

Precondition: `0.0 ≤ p ≤ 1.0`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

binomial_distribution Class

Article • 08/03/2021

Generates a binomial distribution.

Syntax

C++

```
template<class IntType = int>
class binomial_distribution
   {
public:
   // types
   typedef IntType result_type;
   struct param_type;

   // constructors and reset functions
   explicit binomial_distribution(result_type t = 1, double p = 0.5);
   explicit binomial_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type t() const;
   double p() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

IntType

The integer result type, defaults to `int`. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `int` if none is provided, distributed according to the Binomial Distribution discrete probability function. The following table links to articles about individual members.

binomial_distribution
param_type

The property members `t()` and `p()` return the currently stored distribution parameter values *t* and *p* respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the binomial distribution discrete probability function, see the Wolfram MathWorld article Binomial Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int t, const double p, const int& s) {

    // uncomment to use a non-deterministic seed
```

```
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::binomial_distribution<> distr(t, p);

    std::cout << std::endl;
    std::cout << "p == " << distr.p() << std::endl;
    std::cout << "t == " << distr.t() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' <<
std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int    t_dist = 1;
    double p_dist = 0.5;
    int    samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter an integer value for t distribution (where 0 <= t):
";
    std::cin >> t_dist;
    std::cout << "Enter a double value for p distribution (where 0.0 <= p <=
1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(t_dist, p_dist, samples);
}
```

First run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where 0 <= t): 22
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .25
Enter an integer value for a sample count: 100
```

```
p == 0.25
t == 22
Histogram for 100 samples:
    1 :
    2 ::
    3 ::::::::::::
    4 ::::::::::::::
    5 ::::::::::::::::::::::::
    6 ::::::::::::::::::
    7 ::::::::::::
    8 ::::::
    9 ::::::
    11 :
    12 :
```

Second run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where 0 <= t): 22
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .5
Enter an integer value for a sample count: 100

p == 0.5
t == 22
Histogram for 100 samples:
    6 :
    7 ::
    8 :::::::::
    9 ::::::::::
    10 ::::::::::::::::
    11 ::::::::::::::::::
    12 ::::::::::
    13 ::::::::::::
    14 ::::::::::::::
    15 ::
    16 ::
```

Third run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for t distribution (where 0 <= t): 22
Enter a double value for p distribution (where 0.0 <= p <= 1.0): .75
Enter an integer value for a sample count: 100

p == 0.75
t == 22
Histogram for 100 samples:
```

```
13 ::::
14 ::::::::::
15 ::::::::::::::
16 ::::::::::::::::::::
17 ::::::::::::::
18 ::::::::::::::::
19 ::::::::::
20 ::::::
21 :
```

Requirements

Header: <random>

Namespace: std

binomial_distribution::binomial_distribution

Constructs the distribution.

C++

```
explicit binomial_distribution(result_type t = 1, double p = 0.5);
explicit binomial_distribution(const param_type& parm);
```

Parameters

t
The `t` distribution parameter.

p
The `p` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0 ≤ t` and `0.0 ≤ p ≤ 1.0`

The first constructor constructs an object whose stored *p* value holds the value *p* and whose stored *t* value holds the value *t*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

binomial_distribution::param_type

Stores all the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef binomial_distribution<result_type> distribution_type;
   param_type(result_type t = 1, double p = 0.5);
   result_type t() const;
   double p() const;
   .....
   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

t
The `t` distribution parameter.

p
The `p` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0 ≤ t` and `0.0 ≤ p ≤ 1.0`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

cauchy_distribution Class

Article • 08/03/2021

Generates a Cauchy distribution.

Syntax

C++

```
template<class RealType = double>
class cauchy_distribution {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructor and reset functions
   explicit cauchy_distribution(result_type a = 0.0, result_type b = 1.0);
   explicit cauchy_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type a() const;
   result_type b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating-point type, or type `double` if none is provided, distributed according to the Cauchy Distribution. The following table links to articles about individual members.

cauchy_distribution
param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters `a` and `b`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the cauchy distribution, see the Wolfram MathWorld article Cauchy Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;

    std::mt19937 gen(1701);
```

```
    std::cauchy_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'a' distribution
parameter: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'b' distribution
parameter (must be greater than zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

First run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 0
Enter a floating point value for the 'b' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
a() == 0.0000000000
b() == 1.0000000000
Distribution for 10 samples:
    1: -3.4650392984
    2: -2.6369564174
    3: -0.0786978867
    4: -0.0609632093
    5: 0.0589387400
    6: 0.0589539764
    7: 0.1004592006
    8: 1.0965724260
    9: 1.4389408122
    10: 2.5253154706
```

Second run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 0
Enter a floating point value for the 'b' distribution parameter (must be
greater than zero): 10
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
a() == 0.0000000000
b() == 10.0000000000
Distribution for 10 samples:
    1: -34.6503929840
    2: -26.3695641736
    3: -0.7869788674
    4: -0.6096320926
    5: 0.5893873999
    6: 0.5895397637
    7: 1.0045920062
    8: 10.9657242597
    9: 14.3894081218
    10: 25.2531547063
```

Third run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 10
Enter a floating point value for the 'b' distribution parameter (must be
greater than zero): 10
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
a() == 10.0000000000
b() == 10.0000000000
Distribution for 10 samples:
    1: -24.6503929840
    2: -16.3695641736
    3: 9.2130211326
    4: 9.3903679074
    5: 10.5893873999
    6: 10.5895397637
    7: 11.0045920062
    8: 20.9657242597
    9: 24.3894081218
    10: 35.2531547063
```

Requirements

Header: <random>

Namespace: std

cauchy_distribution::cauchy_distribution

Constructs the distribution.

C++

```
explicit cauchy_distribution(result_type a = 0.0, result_type b = 1.0);
explicit cauchy_distribution(const param_type& parm);
```

Parameters

a

The `a` distribution parameter.

b

The `b` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 < b`

The first constructor constructs an object whose stored `a` value holds the value *a* and whose stored `b` value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

cauchy_distribution::param_type

Stores all the parameters of the distribution.

C++

```
struct param_type {
   typedef cauchy_distribution<result_type> distribution_type;
   param_type(result_type a = 0.0, result_type b = 1.0);
   result_type a() const;
   result_type b() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

a
The `a` distribution parameter.

b
The `b` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

chi_squared_distribution Class

Article • 08/03/2021

Generates a chi-squared distribution.

Syntax

C++

```
template<class RealType = double>
class chi_squared_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructor and reset functions
    explicit chi_squared_distribution(RealType n = 1);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating-point type, or type `double` if none is provided, distributed according to the Chi-Squared Distribution. The following table links to articles about individual members.

chi_squared_distribution
param_type

The property function `n()` returns the value for the stored distribution parameter `n`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the chi-squared distribution, see the Wolfram MathWorld article Chi-Squared Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double n, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::chi_squared_distribution<> distr(n);

    std::cout << std::endl;
```

```
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double n_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the \'n\' distribution
parameter (must be greater than zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(n_dist, samples);
}
```

First run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): .5
Enter an integer value for the sample count: 10

min() == 4.94066e-324
max() == 1.79769e+308
n() == 0.5000000000
Distribution for 10 samples:
    1: 0.0007625595
    2: 0.0016895062
```

```
    3: 0.0058683478
    4: 0.0189647765
    5: 0.0556619371
    6: 0.1448191353
    7: 0.1448245325
    8: 0.1903494379
    9: 0.9267525768
    10: 1.5429743723
```

Second run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): .3333
Enter an integer value for the sample count: 10

min() == 4.94066e-324
max() == 1.79769e+308
n() == 0.3333000000
Distribution for 10 samples:
    1: 0.0000148725
    2: 0.0000490528
    3: 0.0003175988
    4: 0.0018454535
    5: 0.0092808795
    6: 0.0389540735
    7: 0.0389562514
    8: 0.0587028468
    9: 0.6183666639
    10: 1.3552086624
```

Third run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): 1000
Enter an integer value for the sample count: 10

min() == 4.94066e-324
max() == 1.79769e+308
n() == 1000.0000000000
Distribution for 10 samples:
    1: 958.5284624473
    2: 958.7882787809
    3: 963.0667684792
    4: 987.9638091514
    5: 1016.2433493745
    6: 1021.9337111110
```

```
7: 1021.9723046240
8: 1035.7622110505
9: 1043.8725156645
10: 1054.7051509381
```

Requirements

Header: <random>

Namespace: std

chi_squared_distribution::chi_squared_distribution

Constructs the distribution.

```
C++
```

```
explicit chi_squared_distribution(result_type n = 1.0);
explicit chi_squared_distribution(const param_type& parm);
```

Parameters

n
The `n` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < n`

The first constructor constructs an object whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

chi_squared_distribution::param_type

Stores the parameters of the distribution.

C++

```
struct param_type {
   typedef chi_squared_distribution<result_type> distribution_type;
   param_type(result_type n = 1.0);
   result_type n() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

n
The `n` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < n`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

discard_block_engine Class

Article • 08/03/2021

Generates a random sequence by discarding values returned by its base engine.

Syntax

```
C++
```

```
template <class Engine, size_t P, size_t R>
class discard_block_engine;
```

Parameters

Engine
The base engine type.

P
Block size. The number of values in each block.

R
Used block. The number of values in each block that are used. The rest are discarded (`P` - `R`). **Precondition**: `0 < R ≤ P`

Members

`discard_block_engine::discard_block_engine`

`discard_block_engine::base`

`discard_block_engine::base_type`

`discard_block_engine::discard`

`discard_block_engine::operator()`

`discard_block_engine::seed`

For more information about engine members, see <random>.

Remarks

This class template describes an engine adaptor that produces values by discarding some of the values returned by its base engine.

Requirements

Header: <random>

Namespace: std

See also

<random>

discrete_distribution Class

Article • 08/03/2021

Generates a discrete integer distribution that has uniform-width intervals with uniform probability in each interval.

Syntax

C++

```
template<class IntType = int>
class discrete_distribution
   {
public:
   // types
   typedef IntType result_type;
   struct param_type;

   // constructor and reset functions
   discrete_distribution();
   template <class InputIterator>
   discrete_distribution(InputIterator firstW, InputIterator lastW);
   discrete_distribution(initializer_list<double> weightlist);
   template <class UnaryOperation>
   discrete_distribution(size_t count, double xmin, double xmax,
UnaryOperation funcweight);
   explicit discrete_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   vector<double> probabilities() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

IntType

The integer result type, defaults to `int`. For possible types, see <random>.

Remarks

This sampling distribution has uniform-width intervals with uniform probability in each interval. For information about other sampling distributions, see piecewise_linear_distribution Class and piecewise_constant_distribution Class.

The following table links to articles about individual members:

discrete_distribution
param_type

The property function `vector<double> probabilities()` returns the individual probabilities for each integer generated.

For more information about distribution classes and their members, see <random>.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    discrete_distribution<> distr({ 1, 2, 3, 4, 5 });

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "probabilities (value: probability):" << endl;
    vector<double> p = distr.probabilities();
    int counter = 0;
    for (const auto& n : p) {
        cout << fixed << setw(11) << counter << ": " << setw(14) <<
setprecision(10) << n << endl;
        ++counter;
    }
```

```
    cout << endl;

    // generate the distribution as a histogram
    map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << s << " samples:" << endl;
    for (const auto& elem : histogram) {
        cout << setw(5) << elem.first << ' ' << string(elem.second, ':') <<
endl;
    }
    cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values."
<< endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for the sample count: 100
min() == 0
max() == 4
probabilities (value: probability):
          0:   0.0666666667
          1:   0.1333333333
          2:   0.2000000000
          3:   0.2666666667
          4:   0.3333333333

Distribution for 100 samples:
    0 :::
    1 ::::::::::::::
    2 ::::::::::::::::::
    3 ::::::::::::::::::::::::::::
    4 ::::::::::::::::::::::::::::::::::
```

Requirements

Header: <random>

Namespace: std

discrete_distribution::discrete_distribution

Constructs the distribution.

```
C++
```

```
// default constructor
discrete_distribution();

// construct using a range of weights, [firstW, lastW)
template <class InputIterator>
discrete_distribution(InputIterator firstW, InputIterator lastW);

// construct using an initializer list for range of weights
discrete_distribution(initializer_list<double> weightlist);

// construct using unary operation function
template <class UnaryOperation>
discrete_distribution(size_t count, double low, double high, UnaryOperation
weightfunc);

// construct from an existing param_type structure
explicit discrete_distribution(const param_type& parm);
```

Parameters

firstW
The first iterator in the list from which to construct the distribution.

lastW
The last iterator in the list from which to construct the distribution (non-inclusive because iterators use an empty element for the end).

weightlist
The initializer_list from which to construct the distribution.

count
The number of elements in the distribution range. If `count==0`, equivalent to the default constructor (always generates zero).

low
The lowest value in the distribution range.

high
The highest value in the distribution range.

weightfunc
The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to `double`.

parm
The `param_type` structure used to construct the distribution.

Remarks

The default constructor constructs an object whose stored probability value has one element with value 1. This will result in a distribution that always generates a zero.

The iterator range constructor that has parameters *firstW* and *lastW* constructs a distribution object by using weight values taken from the iterators over the interval sequence [*firstW*, *lastW*).

The initializer list constructor that has a *weightlist* parameter constructs a distribution object with weights from the initializer list *weightlist*.

The constructor that has *count*, *low*, *high*, and *weightfunc* parameters constructs a distribution object initialized based on these rules:

- If *count* < 1, **n** = 1, and as such is equivalent to the default constructor, always generating zero.
- If *count* > 0, **n** = *count*. Provided **d** = (*high* - *low*) / **n** is greater than zero, using **d** uniform subranges, each weight is assigned as follows: `weight[k] = weightfunc(x)`, where **x** = *low* + **k** * **d** + **d** / 2, for **k** = 0, ..., **n** - 1.

The constructor that has a `param_type` parameter *parm* constructs a distribution object using *parm* as the stored parameter structure.

discrete_distribution::param_type

Stores all the parameters of the distribution.

C++

```
struct param_type {
   typedef discrete_distribution<result_type> distribution_type;
   param_type();
```

```
    // construct using a range of weights, [firstW, lastW)
    template <class InputIterator>
    param_type(InputIterator firstW, InputIterator lastW);

    // construct using an initializer list for range of weights
    param_type(initializer_list<double> weightlist);

    // construct using unary operation function
    template <class UnaryOperation>
    param_type(size_t count, double low, double high, UnaryOperation
weightfunc);

    std::vector<double> probabilities() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
    };
```

Parameters

firstW
The first iterator in the list from which to construct the distribution.

lastW
The last iterator in the list from which to construct the distribution (non-inclusive because iterators use an empty element for the end).

weightlist
The initializer_list from which to construct the distribution.

count
The number of elements in the distribution range. If *count* is 0, this is equivalent to the default constructor (always generates zero).

low
The lowest value in the distribution range.

high
The highest value in the distribution range.

weightfunc
The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to `double`.

right
The `param_type` object to compare to this.

Remarks

This parameter package can be passed to `operator()` to generate the return value.

See also

<random>

exponential_distribution Class

Article • 08/03/2021

Generates an exponential distribution.

Syntax

C++

```
template<class RealType = double>
class exponential_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructors and reset functions
   explicit exponential_distribution(result_type lambda = 1.0);
   explicit exponential_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type lambda() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

URNG

The random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `double` if none is provided, distributed according to the Exponential Distribution. The following table links to articles about individual members.

exponential_distribution
param_type

The property member function `lambda()` returns the value for the stored distribution parameter `lambda`.

The property member function `param()` sets or returns the `param_type` stored distribution parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the exponential distribution, see the Wolfram MathWorld article Exponential Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double l, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::exponential_distribution<> distr(l);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "lambda() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.lambda() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }
```

```
    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double l_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'lambda' distribution
parameter (must be greater than zero): ";
    std::cin >> l_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(l_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'lambda' distribution parameter (must
be greater than zero): 1
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
lambda() == 1.0000000000
Distribution for 10 samples:
    1: 0.0936880533
    2: 0.1225944894
    3: 0.6443593183
    4: 0.6551171649
    5: 0.7313457551
    6: 0.7313557977
    7: 0.7590097389
    8: 1.4466885214
    9: 1.6434088411
    10: 2.1201210996
```

Requirements

Header: <random>

Namespace: std

exponential_distribution::exponential_distribution

Constructs the distribution.

```
C++
```

```
explicit exponential_distribution(result_type lambda = 1.0);
explicit exponential_distribution(const param_type& parm);
```

Parameters

lambda
The `lambda` distribution parameter.

parm
The parameter package used to construct the distribution.

Remarks

Precondition: `0.0 < lambda`

The first constructor constructs an object whose stored `lambda` value holds the value *lambda*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

exponential_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef exponential_distribution<result_type> distribution_type;
   param_type(result_type lambda = 1.0);
   result_type lambda() const;
```

```
    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
    };
```

Parameters

lambda
The `lambda` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < lambda`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

extreme_value_distribution Class

Article • 08/03/2021

Generates an extreme value distribution.

Syntax

C++

```
template<class RealType = double>
class extreme_value_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructor and reset functions
   explicit extreme_value_distribution(result_type a = 0.0, result_type b = 1.0);
   explicit extreme_value_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type a() const;
   result_type b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to **`double`**. For possible types, see <random>.

URNG

The random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating-point type, or type `double` if none is provided, distributed according to the Extreme Value Distribution. The following table links to articles about individual members.

extreme_value_distribution
param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters `a` and `b`.

For more information about distribution classes and their members, see <random>.

For detailed information about the extreme value distribution, see the Wolfram MathWorld article Extreme Value Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;

    std::mt19937 gen(1701);

    std::extreme_value_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) << std::setprecision(10)
<< distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) << std::setprecision(10)
<< distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
```

```
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default values."
<< std::endl;
    std::cout << "Enter a floating point value for the \'a\' distribution
parameter: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the \'b\' distribution
parameter (must be greater than zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter: 0
Enter a floating point value for the 'b' distribution parameter (must be greater
than zero): 1
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
a() == 0.0000000000
b() == 1.0000000000
Distribution for 10 samples:
    1: -0.8813940331
    2: -0.7698972281
    3: 0.2951258007
    4: 0.3110450734
    5: 0.4210546820
    6: 0.4210688771
    7: 0.4598857960
    8: 1.3155194200
    9: 1.5379170046
    10: 2.0568757061
```

Requirements

Header: <random>

Namespace: std

extreme_value_distribution::extreme_value_distribution

Constructs the distribution.

```
C++
```

```
explicit extreme_value_distribution(result_type a_value = 0.0, result_type b_value
= 1.0);
explicit extreme_value_distribution(const param_type& parm);
```

Parameters

a_value
The `a` distribution parameter.

b_value
The `b` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 < b`

The first constructor constructs an object whose stored `a` value holds the value *a_value* and whose stored `b` value holds the value *b_value*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

extreme_value_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef extreme_value_distribution<result_type> distribution_type;
   param_type(result_type a_value = 0.0, result_type b_value = 1.0);
   result_type a() const;
   result_type b() const;
```

```
    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
    };
```

Parameters

a_value
The `a` distribution parameter.

b_value
The `b` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

fisher_f_distribution Class

Article • 08/03/2021

Generates a Fisher F distribution.

Syntax

C++

```
template<class RealType = double>
class fisher_f_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;  // constructor and reset functions
   explicit fisher_f_distribution(result_type m = 1.0, result_type n = 1.0);
   explicit fisher_f_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type m() const;
   result_type n() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to **`double`**. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating-point type, or type `double` if none is provided, distributed according to the Fisher's F-Distribution. The following table links to articles about individual members.

fisher_f_distribution
param_type

The property functions `m()` and `n()` return the values for the stored distribution parameters `m` and `n` respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the F-distribution, see the Wolfram MathWorld article F-Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double m, const double n, const int s) {

    // uncomment to use a non-deterministic seed
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1701);
```

```
    std::fisher_f_distribution<> distr(m, n);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "m() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.m() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double m_dist = 1;
    double n_dist = 1;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the \'m\' distribution
parameter (must be greater than zero): ";
    std::cin >> m_dist;
    std::cout << "Enter a floating point value for the \'n\' distribution
parameter (must be greater than zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(m_dist, n_dist, samples);
}
```

Output

First run:

```
Output
```

```
Enter a floating point value for the 'm' distribution parameter (must be
greater than zero): 1
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
m() == 1.0000000000
n() == 1.0000000000
Distribution for 10 samples:
    1: 0.0204569549
    2: 0.0221376644
    3: 0.0297234962
    4: 0.1600937252
    5: 0.2775342196
    6: 0.3950701700
    7: 0.8363200295
    8: 0.9512500702
    9: 2.7844815974
    10: 3.4320929653
```

Second run:

Output

```
Enter a floating point value for the 'm' distribution parameter (must be
greater than zero): 1
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): .1
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
m() == 1.0000000000
n() == 0.1000000000
Distribution for 10 samples:
    1: 0.0977725649
    2: 0.5304122767
    3: 4.9468518084
    4: 25.1012074939
    5: 48.8082121613
    6: 401.8075539377
    7: 8199.5947873699
    8: 226492.6855335717
    9: 2782062.6639740225
    10: 20829747131.7185860000
```

Third run:

Output

```
Enter a floating point value for the 'm' distribution parameter (must be
greater than zero): .1
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
m() == 0.1000000000
n() == 1.0000000000
Distribution for 10 samples:
    1: 0.0000000000
    2: 0.0000000000
    3: 0.0000000000
    4: 0.0000000000
    5: 0.0000000033
    6: 0.0000073975
    7: 0.0000703800
    8: 0.0280427735
    9: 0.2660239949
    10: 3.4363333954
```

Requirements

Header: <random>

Namespace: std

fisher_f_distribution::fisher_f_distribution

Constructs the distribution.

C++

```
explicit fisher_f_distribution(result_type m = 1.0, result_type n = 1.0);
explicit fisher_f_distribution(const param_type& parm);
```

Parameters

m

The `m` distribution parameter.

n

The `n` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 < m` and `0.0 < n`

The first constructor constructs an object whose stored `m` value holds the value *m* and whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

fisher_f_distribution::param_type

Stores the parameters of the distribution.

C++

```
struct param_type {
   typedef fisher_f_distribution<result_type> distribution_type;
   param_type(result_type m = 1.0, result_type n = 1.0);
   result_type m() const;
   result_type n() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

m
The `m` distribution parameter.

n
The `n` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < m` and `0.0 < n`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

gamma_distribution Class

Article • 08/03/2021

Generates a gamma distribution.

Syntax

C++

```
template<class RealType = double>
class gamma_distribution {
public:
    // types
    typedef RealType result_type;
    struct param_type;

    // constructors and reset functions
    explicit gamma_distribution(result_type alpha = 1.0, result_type beta =
1.0);
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type alpha() const;
    result_type beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating-point type, or type `double` if none is provided, distributed according to the Gamma Distribution. The following table links to articles about individual members.

gamma_distribution
param_type

The property functions `alpha()` and `beta()` return their respective values for stored distribution parameters *alpha* and *beta*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the gamma distribution, see the Wolfram MathWorld article Gamma Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
```

```
    std::mt19937 gen(1701);

    std::gamma_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "alpha() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.alpha() << std::endl;
    std::cout << "beta() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.beta() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'alpha' distribution
parameter (must be greater than zero): ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'beta' distribution
parameter (must be greater than zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'alpha' distribution parameter (must be
greater than zero): 1
Enter a floating point value for the 'beta' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == 4.94066e-324
max() == 1.79769e+308
alpha() == 1.0000000000
beta() == 1.0000000000
Distribution for 10 samples:
    1: 0.0936880533
    2: 0.1225944894
    3: 0.6443593183
    4: 0.6551171649
    5: 0.7313457551
    6: 0.7313557977
    7: 0.7590097389
    8: 1.4466885214
    9: 1.6434088411
    10: 2.1201210996
```

Requirements

Header: <random>

Namespace: std

gamma_distribution::gamma_distribution

Constructs the distribution.

C++

```
explicit gamma_distribution(result_type alpha = 1.0, result_type beta =
1.0);
explicit gamma_distribution(const param_type& parm);
```

Parameters

alpha

The `alpha` distribution parameter.

beta
The `beta` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < alpha` and `0.0 < beta`

The first constructor constructs an object whose stored `alpha` value holds the value *alpha* and whose stored `beta` value holds the value *beta*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

gamma_distribution::param_type

Stores the parameters of the distribution.

C++

```
struct param_type {
   typedef gamma_distribution<result_type> distribution_type;
   param_type(result_type alpha = 1.0, result_type beta 1.0);
   result_type alpha() const;
   result_type beta() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

alpha
The `alpha` distribution parameter.

beta
The `beta` distribution parameter.

right
The `param_type` instance to compare this to.

Remarks

Precondition: `0.0 < alpha` and `0.0 < beta`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

geometric_distribution Class

Article • 08/03/2021

Generates a geometric distribution.

Syntax

C++

```
template<class IntType = int>
class geometric_distribution {
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructors and reset functions
    explicit geometric_distribution(double p = 0.5);
    explicit geometric_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class URNG>
    result_type operator()(URNG& gen);
    template <class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType

The integer result type, defaults to `int`. For possible types, see <random>.

URNG

The uniform random number generator engine. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type with a geometric distribution. The following table links to articles about individual members.

geometric_distribution
param_type

The property function `p()` returns the value for stored distribution parameter `p`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the chi-squared distribution, see the Wolfram MathWorld article Geometric Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::geometric_distribution<> distr(p);

    std::cout << std::endl;
```

```
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "p() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' <<
std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 0.5;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the \'p\' distribution
parameter: ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```

First test:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'p' distribution parameter: .5
Enter an integer value for the sample count: 100

min() == 0
max() == 2147483647
p() == 0.5000000000
Distribution for 100 samples:
    0 ::::::::::::::::::::::::::::::::::::::::::::::::::
    1 ::::::::::::::::::::::::
    2 ::::::::::::
    3 ::::::
```

```
    4 ::
    5 :
```

Second test:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'p' distribution parameter: .1
Enter an integer value for the sample count: 100

min() == 0
max() == 2147483647
p() == 0.1000000000
Distribution for 100 samples:
    0 :::::::::
    1 ::::::::::
    2 ::::::::::
    3 :::::::
    4 :::::
    5 ::::::::
    6 :::
    7 ::::::
    8 :::::::
    9 :::::
   10 :::
   11 :::
   12 ::
   13 :
   14 :::
   15 ::
   16 :::
   17 :::
   20 :::::
   21 :
   29 :
   32 :
   35 :
```

Requirements

Header: <random>

Namespace: std

geometric_distribution::geometric_distribution

Constructs the distribution.

C++

```
explicit geometric_distribution(double p = 0.5);
explicit geometric_distribution(const param_type& parm);
```

Parameters

p
The `p` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < p && p < 1.0`

The first constructor constructs an object whose stored `p` value holds the value *p*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

geometric_distribution::param_type

Stores the parameters of the distribution.

C++

```
struct param_type {
   typedef geometric_distribution<result_type> distribution_type;
   param_type(double p = 0.5);
   double p() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

p
The `p` distribution parameter.

right
The `param_type` instance to compare this to.

Remarks

Precondition: `0.0 < p && p < 1.0`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

independent_bits_engine Class

Article • 08/03/2021

Generates a random sequence of numbers with a specified number of bits by repacking bits from the values returned by its base engine.

Syntax

```
C++
```

```
template <class Engine, size_t W, class UIntType>
class independent_bits_engine;
```

Parameters

Engine
The base engine type.

W
Word size. Size, in bits, of each number generated. **Precondition**: `0 < W ≤ numeric_limits<UIntType>::digits`

UIntType
The unsigned integer result type. For possible types, see <random>.

Members

`independent_bits_engine::independent_bits_engine`

`independent_bits_engine::base`

`independent_bits_engine::base_type`

`independent_bits_engine::discard`

`independent_bits_engine::operator()`

`independent_bits_engine::seed`

For more information about engine members, see <random>.

Remarks

This class template describes an *engine adaptor* that produces values by repacking bits from the values returned by its base engine, resulting in *W*-bit values.

Requirements

Header: <random>

Namespace: std

See also

<random>

linear_congruential_engine Class

Article • 08/03/2021

Generates a random sequence by the linear congruential algorithm.

Syntax

C++

```
class linear_congruential_engine{
   public:  // types
   typedef UIntType result_type;
   // engine characteristics
   static constexpr result_type multiplier = a;
   static constexpr result_type increment = c;
   static constexpr result_type modulus = m;
   static constexpr result_type min() { return c == 0u  1u: 0u; }
   static constexpr result_type max() { return m - 1u; }
   static constexpr result_type default_seed = 1u;
   // constructors and seeding functions
   explicit linear_congruential_engine(result_type s = default_seed);
   template <class Sseq>
   explicit linear_congruential_engine(Sseq& q);
   void seed(result_type s = default_seed);
   template <class Sseq>
   void seed(Sseq& q);
   // generating functions
   result_type operator()();
   void discard(unsigned long long z);
   };
```

Parameters

UIntType

The unsigned integer result type. For possible types, see <random>.

A

Multiplier. **Precondition**: See Remarks section.

C

Increment. **Precondition**: See Remarks section.

M

Modulus. **Precondition**: See remarks.

Members

`linear_congruential_engine::linear_congruential_engine`

`linear_congruential_engine::discard`

`linear_congruential_engine::max`

`linear_congruential_engine::min`

`linear_congruential_engine::operator()`

`linear_congruential_engine::seed`

`default_seed` is a member constant, defined as `1u`, used as the default parameter value for `linear_congruential_engine::seed` and the single value constructor.

For more information about engine members, see <random>.

Remarks

The `linear_congruential_engine` class template is the simplest generator engine, but not the fastest or highest quality. An improvement over this engine is the substract_with_carry_engine. Neither of these engines is as fast or with as high quality results as the mersenne_twister_engine.

This engine produces values of a user-specified unsigned integral type using the recurrence relation (*period*) `x(i) = (A * x(i-1) + C) mod M`.

If *M* is zero, the value used for this modulus operation is `numeric_limits<result_type>::max() + 1`. The engine's state is the last value returned, or the seed value if no call has been made to `operator()`.

If *M* is not zero, the values of the template arguments *A* and *C* must be less than *M*.

Although you can construct a generator from this engine directly, you can also use one of these predefined typedefs.

`minstd_rand0`: 1988 minimal standard engine (Lewis, Goodman, and Miller, 1969).

```
C++
```

```
typedef linear_congruential_engine<unsigned int, 16807, 0, 2147483647>
minstd_rand0;
```

`minstd_rand`: Updated minimal standard engine `minstd_rand0` (Park, Miller, and Stockmeyer, 1993).

C++

```
typedef linear_congruential_engine<unsigned int, 48271, 0, 2147483647>
minstd_rand;
```

For detailed information about the linear congruential engine algorithm, see the Wikipedia article Linear congruential generator.

Requirements

Header: <random>

Namespace: std

See also

<random>

lognormal_distribution Class

Article • 08/03/2021

Generates a log normal distribution.

Syntax

C++

```
template <class RealType = double>
class lognormal_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;
   // constructor and reset functions
   explicit lognormal_distribution(result_type m = 0.0, result_type s =
1.0);
   explicit lognormal_distribution(const param_type& parm);
   void reset();
   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);
   // property functions
   result_type m() const;
   result_type s() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `double` if none is provided, distributed according to the Log

Normal Distribution. The following table links to articles about individual members.

lognormal_distribution
param_type

The property functions `m()` and `s()` return the values for the stored distribution parameters *m* and *s*, respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the LogNormal distribution, see the Wolfram MathWorld article LogNormal Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const double m, const double s, const int samples) {

    // uncomment to use a non-deterministic seed
    //    random_device gen;
    //    mt19937 gen(rd());
    mt19937 gen(1701);

    lognormal_distribution<> distr(m, s);
```

```
    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "m() == " << fixed << setw(11) << setprecision(10) << distr.m()
<< endl;
    cout << "s() == " << fixed << setw(11) << setprecision(10) << distr.s()
<< endl;

    // generate the distribution as a histogram
    map<double, int> histogram;
    for (int i = 0; i < samples; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << samples << " samples:" << endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        cout << fixed << setw(11) << ++counter << ": "
            << setw(14) << setprecision(10) << elem.first << endl;
    }
    cout << endl;
}

int main()
{
    double m_dist = 1;
    double s_dist = 1;
    int samples = 10;

    cout << "Use CTRL-Z to bypass data entry and run using default values."
<< endl;
    cout << "Enter a floating point value for the 'm' distribution
parameter: ";
    cin >> m_dist;
    cout << "Enter a floating point value for the 's' distribution parameter
(must be greater than zero): ";
    cin >> s_dist;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(m_dist, s_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'm' distribution parameter: 0
Enter a floating point value for the 's' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10
```

```
min() == -1.79769e+308
max() == 1.79769e+308
m() == 0.0000000000
s() == 1.0000000000
Distribution for 10 samples:
    1: 0.3862809339
    2: 0.4128865601
    3: 0.4490576787
    4: 0.4862035428
    5: 0.5930607126
    6: 0.8190778771
    7: 0.8902379317
    8: 2.8332911667
    9: 5.1359445565
    10: 5.4406507912
```

Requirements

Header: <random>

Namespace: std

lognormal_distribution::lognormal_distribution

Constructs the distribution.

C++

```
explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
explicit lognormal_distribution(const param_type& parm);
```

Parameters

m

The `m` distribution parameter.

s

The `s` distribution parameter.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 < s`

The first constructor constructs an object whose stored `m` value holds the value *m* and whose stored `s` value holds the value *s*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

lognormal_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef lognormal_distribution<result_type> distribution_type;
   param_type(result_type m = 0.0, result_type s = 1.0);
   result_type m() const;
   result_type s() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
};
```

Parameters

m
The `m` distribution parameter.

s
The `s` distribution parameter.

right
The `param_type` structure used to compare.

Remarks

Precondition: `0.0 < s`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

mersenne_twister_engine Class

Article • 08/03/2021

Generates a high quality random sequence of integers based on the Mersenne twister algorithm.

Syntax

```
C++
```

```
template <class UIntType,
    size_t W, size_t N, size_t M, size_t R,
    UIntType A, size_t U, UIntType D, size_t S,
    UIntType B, size_t T, UIntType C, size_t L, UIntType F>
class mersenne_twister_engine;
```

Parameters

UIntType
The unsigned integer result type. For possible types, see <random>.

W
Word size. Size of each word, in bits, of the state sequence. **Precondition**: `2u < W ≤ numeric_limits<UIntType>::digits`

N
State size. The number of elements (values) in the state sequence.

M
Shift size. The number of elements to skip during each twist. **Precondition**: `0 < M ≤ N`

R
Mask bits. **Precondition**: `R ≤ W`

A
XOR mask. **Precondition**: `A ≤ (1u<<W) - 1u`

U, *S*, *T*, *L*
Tempering shift parameters. Used as shift values during scrambling (tempering).
Precondition: `U,S,T,L ≤ W`

D, *B*, *C*

Tempering bit mask parameters. Used as bit mask values during scrambling (tempering). Precondition: `D,B,C ≤ (1u<<W) - 1u`

F

Initialization multiplier. Used to help with initialization of the sequence. Precondition: `F ≤ (1u<<W) - 1u`

Members

`mersenne_twister_engine::mersenne_twister_engine`

`mersenne_twister_engine::discard`

`mersenne_twister_engine::max`

`mersenne_twister_engine::min`

`mersenne_twister_engine::operator()`

`mersenne_twister_engine::seed`

`default_seed` is a member constant, defined as `5489u`, used as the default parameter value for `mersenne_twister_engine::seed` and the single value constructor.

For more information about engine members, see <random>.

Remarks

This class template describes a random number engine, returning values on the closed interval [`0`, 2^W - `1`]. It holds a large integral value with `W * (N - 1) + R` bits. It extracts *W* bits at a time from this large value, and when it has used all the bits it twists the large value by shifting and mixing the bits so that it has a new set of bits to extract from. The engine's state is the last `N` `W`-bit values used if `operator()` has been called at least *N* times, otherwise the `M` `W`-bit values that have been used and the last `N - M` values of the seed.

The generator twists the large value that it holds by using a twisted generalized feedback shift register defined by shift values *N* and *M*, a twist value *R*, and a conditional XOR-mask *A*. Additionally, the bits of the raw shift register are scrambled (tempered) according to a bit-scrambling matrix defined by values *U*, *D*, *S*, *B*, *T*, *C*, and *L*.

The template argument `UIntType` must be large enough to hold values up to 2^W - `1`. The values of the other template arguments must satisfy the following requirements: `2u < W, 0 < M, M ≤ N, R ≤ W, U ≤ W, S ≤ W, T ≤ W, L ≤ W, W ≤`

`numeric_limits<UIntType>::digits, A ≤ (1u<<W) - 1u, B ≤ (1u<<W) - 1u, C ≤ (1u<<W) - 1u, D ≤ (1u<<W) - 1u, and F ≤ (1u<<W) - 1u`.

Although you can construct a generator from this engine directly, it is recommended you use one of these predefined typedefs:

`mt19937`: 32-bit Mersenne twister engine (Matsumoto and Nishimura, 1998).

C++

```
typedef mersenne_twister_engine<unsigned int, 32, 624, 397,
    31, 0x9908b0df,
    11, 0xffffffff,
    7, 0x9d2c5680,
    15, 0xefc60000,
    18, 1812433253> mt19937;
```

`mt19937_64`: 64-bit Mersenne twister engine (Matsumoto and Nishimura, 2000).

C++

```
typedef mersenne_twister_engine<unsigned long long, 64, 312, 156,
    31, 0xb5026f5aa96619e9ULL,
    29, 0x5555555555555555ULL,
    17, 0x71d67fffeda60000ULL,
    37, 0xfff7eee000000000ULL,
    43, 6364136223846793005ULL> mt19937_64;
```

For detailed information about the Mersenne twister algorithm, see the Wikipedia article Mersenne twister.

Example

For a code example, see <random>.

Requirements

Header: <random>

Namespace: std

See also

<random>

negative_binomial_distribution Class

Article • 08/03/2021

Generates a negative binomial distribution.

Syntax

```
template<class IntType = int>
class negative_binomial_distribution
{
public:
    // types
    typedef IntType result_type;
    struct param_type;

    // constructor and reset functions
    explicit negative_binomial_distribution(result_type k = 1, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template `<`class URNG>
    result_type operator()(URNG& gen);
    template `<`class URNG>
    result_type operator()(URNG& gen, const param_type& parm);

    // property functions
    result_type k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Parameters

IntType
The integer result type, defaults to `int`. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `int` if none is provided, distributed according to the Negative Binomial Distribution discrete probability function. The following table links to articles about individual members.

negative_binomial_distribution
param_type

The property members `k()` and `p()` return the currently stored distribution parameter values *k* and *p* respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the negative binomial distribution discrete probability function, see the Wolfram MathWorld article Negative Binomial Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int k, const double p, const int& s) {

    // uncomment to use a non-deterministic seed
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::negative_binomial_distribution<> distr(k, p);

    std::cout << std::endl;
    std::cout << "k == " << distr.k() << std::endl;
    std::cout << "p == " << distr.p() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Histogram for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' << std::string(elem.second, ':') <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int    k_dist = 1;
    double p_dist = 0.5;
    int    samples = 100;
```

```
    std::cout << "Use CTRL-Z to bypass data entry and run using default values." << std::endl;
    std::cout << "Enter an integer value for k distribution (where 0 < k): ";
    std::cin >> k_dist;
    std::cout << "Enter a double value for p distribution (where 0.0 < p <= 1.0): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for a sample count: ";
    std::cin >> samples;

    test(k_dist, p_dist, samples);
}
```

First run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for k distribution (where 0 `<` k): 1
Enter a double value for p distribution (where 0.0 `<`p `<`= 1.0): .5
Enter an integer value for a sample count: 100

k == 1
p == 0.5
Histogram for 100 samples:
    0 ::::::::::::::::::::::::::::::::::::::::
    1 ::::::::::::::::::::::::::::::
    2 ::::::::::::
    3 :::::::
    4 ::::
    5 ::
```

Second run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for k distribution (where 0 `<` k): 100
Enter a double value for p distribution (where 0.0 `<` p <= 1.0): .667
Enter an integer value for a sample count: 100

k == 100
p == 0.667
Histogram for 100 samples:
    31 ::
    32 :
    33 ::
    34 :
    35 ::
    37 ::
    38 :
    39 :
    40 ::
    41 :::
    42 :::
    43 :::::
    44 :::::
    45 ::::
    46 ::::::
    47 ::::::::
    48 :::
    49 :::
    50 :::::::::
    51 :::::::
    52 ::
```

```
    53 :::
    54 :::::
    56 ::::
    58 :
    59 :::::
    60 ::
    61 :
    62 ::
    64 :
    69 ::::
```

Requirements

Header: <random>

Namespace: std

negative_binomial_distribution::negative_binomial_distribution

Constructs the distribution.

```
C++
```

```
explicit negative_binomial_distribution(result_type k = 1, double p = 0.5);
explicit negative_binomial_distribution(const param_type& parm);
```

Parameters

k
The `k` distribution parameter.

p
The `p` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < k` and `0.0 < p ≤ 1.0`

The first constructor constructs an object whose stored `p` value holds the value *p* and whose stored `k` value holds the value *k*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

negative_binomial_distribution::param_type

Stores the parameters of the distribution.

struct param_type { typedef negative_binomial_distribution<result_type> distribution_type;
param_type(result_type k = 1, double p = 0.5); result_type k() const; double p() const;

bool operator==(const param_type& right) const; bool operator!=(const param_type& right) const; };

Parameters

k
The `k` distribution parameter.

p
The `p` distribution parameter.

right
The `param_type` structure used to compare.

Remarks

Precondition: `0.0 < k` and `0.0 < p ≤ 1.0`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

normal_distribution Class

Article • 08/03/2021

Generates a normal distribution.

Syntax

C++

```
template<class RealType = double>
class normal_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructors and reset functions
   explicit normal_distribution(result_type mean = 0.0, result_type stddev =
1.0);
   explicit normal_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type mean() const;
   result_type stddev() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to **`double`**. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `double` if none is provided, distributed according to the Normal Distribution. The following table links to articles about individual members.

normal_distribution
param_type

The property functions `mean()` and `stddev()` return the values for the stored distribution parameters *mean* and *stddev* respectively.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the Normal distribution, see the Wolfram MathWorld article Normal Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const double m, const double s, const int samples) {

    // uncomment to use a non-deterministic seed
    //    random_device gen;
    //    mt19937 gen(rd());
    mt19937 gen(1701);
```

```
    normal_distribution<> distr(m, s);

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "m() == " << fixed << setw(11) << setprecision(10) <<
distr.mean() << endl;
    cout << "s() == " << fixed << setw(11) << setprecision(10) <<
distr.stddev() << endl;

    // generate the distribution as a histogram
    map<double, int> histogram;
    for (int i = 0; i < samples; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << samples << " samples:" << endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        cout << fixed << setw(11) << ++counter << ": "
            << setw(14) << setprecision(10) << elem.first << endl;
    }
    cout << endl;
}

int main()
{
    double m_dist = 1;
    double s_dist = 1;
    int samples = 10;

    cout << "Use CTRL-Z to bypass data entry and run using default values."
<< endl;
    cout << "Enter a floating point value for the 'mean' distribution
parameter: ";
    cin >> m_dist;
    cout << "Enter a floating point value for the 'stddev' distribution
parameter (must be greater than zero): ";
    cin >> s_dist;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(m_dist, s_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter: 0
Enter a floating point value for the 'stddev' distribution parameter (must
be greater than zero): 1
```

```
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
m() == 0.0000000000
s() == 1.0000000000
Distribution for 10 samples:
    1: -0.8845823965
    2: -0.1995761116
    3: -0.1162665130
    4: -0.0685154932
    5: 0.0403741461
    6: 0.1591327792
    7: 1.0414389924
    8: 1.5876269426
    9: 1.6362637713
    10: 2.7821317338
```

Requirements

Header: <random>

Namespace: std

normal_distribution::normal_distribution

Constructs the distribution.

```
C++
```

```
explicit normal_distribution(result_type mean = 0.0, result_type stddev =
1.0);
explicit normal_distribution(const param_type& parm);
```

Parameters

mean
The `mean` distribution parameter.

stddev
The `stddev` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < stddev`

The first constructor constructs an object whose stored `mean` value holds the value *mean* and whose stored `stddev` value holds the value *stddev*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

normal_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef normal_distribution<result_type> distribution_type;
   param_type(result_type mean = 0.0, result_type stddev = 1.0);
   result_type mean() const;
   result_type stddev() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

mean
The `mean` distribution parameter.

stddev
The `stddev` distribution parameter.

right
The `param_type` structure used to compare.

Remarks

Precondition: `0.0 < stddev`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

piecewise_constant_distribution Class

Article • 08/03/2021

Generates a piecewise constant distribution that has varying-width intervals with uniform probability in each interval.

Syntax

```
C++
```

```
template<class RealType = double>
class piecewise_constant_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructor and reset functions
   piecewise_constant_distribution();
   template <class InputIteratorI, class InputIteratorW>
   piecewise_constant_distribution(
       InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);
   template <class UnaryOperation>
   piecewise_constant_distribution(
      initializer_list<result_type> intervals, UnaryOperation weightfunc);
   template <class UnaryOperation>
   piecewise_constant_distribution(
      size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
   explicit piecewise_constant_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   vector<result_type> intervals() const;
   vector<result_type> densities() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType
The floating point result type, defaults to **`double`**. For possible types, see <random>.

Remarks

This sampling distribution has varying-width intervals with uniform probability in each interval. For information about other sampling distributions, see piecewise_linear_distribution Class and discrete_distribution.

The following table links to articles about individual members:

piecewise_constant_distribution
param_type

The property function `intervals()` returns a `vector<result_type>` with the set of stored intervals of the distribution.

The property function `densities()` returns a `vector<result_type>` with the stored densities for each interval set, which are calculated according to the weights provided in the constructor parameters.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    // Three intervals, non-uniform: 0 to 1, 1 to 6, and 6 to 15
    vector<double> intervals{ 0, 1, 6, 15 };
    // weights determine the densities used by the distribution
    vector<double> weights{ 1, 5, 10 };

    piecewise_constant_distribution<double> distr(intervals.begin(), intervals.end(),
weights.begin());

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "intervals (index: interval):" << endl;
    vector<double> i = distr.intervals();
    int counter = 0;
    for (const auto& n : i) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
        ++counter;
    }
    cout << endl;
```

```
    cout << "densities (index: density):" << endl;
    vector<double> d = distr.densities();
    counter = 0;
    for (const auto& n : d) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) << n << endl;
        ++counter;
    }
    cout << endl;

    // generate the distribution as a histogram
    map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << s << " samples:" << endl;
    for (const auto& elem : histogram) {
        cout << setw(5) << elem.first << '-' << elem.first+1 << ' ' << string(elem.second, ':') <<
endl;
    }
    cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for the sample count: 100
min() == 0
max() == 15
intervals (index: interval):
          0:   0.0000000000
          1:   1.0000000000
          2:   6.0000000000
          3:  15.0000000000
densities (index: density):
          0:   0.0625000000
          1:   0.0625000000
          2:   0.0694444444
Distribution for 100 samples:
    0-1 :::::::
    1-2 ::::::
    2-3 :::::
    3-4 ::::::
    4-5 :::::::
    5-6 ::::::
    6-7 :::
    7-8 ::::::::::
    8-9 ::::::
    9-10 ::::::::::::
    10-11 :::::
    11-12 ::::::
    12-13 :::::::::
    13-14 ::::
    14-15 ::::::::
```

Requirements

Header: <random>

Namespace: std

piecewise_constant_distribution::piecewise_constant_distribution

Constructs the distribution.

```
C++
```

```
// default constructor
piecewise_constant_distribution();

// constructs using a range of intervals, [firstI, lastI), with
// matching weights starting at firstW
template <class InputIteratorI, class InputIteratorW>
piecewise_constant_distribution(InputIteratorI firstI, InputIteratorI lastI, InputIteratorW
firstW);

// constructs using an initializer list for range of intervals,
// with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType>
intervals, UnaryOperation weightfunc);

// constructs using an initializer list for range of count intervals,
// distributed uniformly over [xmin,xmax] with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_constant_distribution(size_t count, RealType xmin, RealType xmax, UnaryOperation
weightfunc);

// constructs from an existing param_type structure
explicit piecewise_constant_distribution(const param_type& parm);
```

Parameters

firstI
An input iterator of the first element in the distribution range.

lastI
An input iterator of the last element in the distribution range.

firstW
An input iterator of the first element in the weights range.

intervals
An initializer_list with the intervals of the distribution.

count
The number of elements in the distribution range.

xmin
The lowest value in the distribution range.

xmax
The highest value in the distribution range. Must be greater than *xmin*.

weightfunc
The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to `double`.

parm
The parameter structure used to construct the distribution.

Remarks

The default constructor sets the stored parameters such that there is one interval, 0 to 1, with a probability density of 1.

The iterator range constructor

```
C++

template <class InputIteratorI, class InputIteratorW>
piecewise_constant_distribution(InputIteratorI firstI, InputIteratorI lastI,
    InputIteratorW firstW);
```

constructs a distribution object with itnervals from iterators over the sequence [`firstI`, `lastI`) and a matching weight sequence starting at `firstW`.

The initializer list constructor

```
C++

template <class UnaryOperation>
piecewise_constant_distribution(initializer_list<result_type>
intervals,
    UnaryOperation weightfunc);
```

constructs a distribution object with intervals from the initializer list *intervals* and weights generated from the function *weightfunc*.

The constructor defined as

```
C++

template <class UnaryOperation>
piecewise_constant_distribution(size_t count, result_type xmin, result_type xmax,
    UnaryOperation weightfunc);
```

constructs a distribution object with *count* intervals distributed uniformly over [`xmin,xmax`], assigning each interval weights according to the function *weightfunc*, and *weightfunc* must accept one parameter and have a return value, both of which are convertible to `double`. **Precondition:** `xmin < xmax`

The constructor defined as

```
C++

explicit piecewise_constant_distribution(const param_type& parm);
```

constructs a distribution object using *parm* as the stored parameter structure.

piecewise_constant_distribution::param_type

Stores all the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef piecewise_constant_distribution<result_type> distribution_type;
   param_type();
   template <class IterI, class IterW>
   param_type(IterI firstI, IterI lastI, IterW firstW);
   template <class UnaryOperation>
   param_type(size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
   std::vector<result_type> densities() const;
   std::vector<result_type> intervals() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

See the constructor parameters for the piecewise_constant_distribution.

Remarks

Precondition: `xmin < xmax`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>
piecewise_linear_distribution

piecewise_linear_distribution Class

Article • 08/03/2021

Generates a piecewise linear distribution that has varying-width intervals with linearly varying probability in each interval.

Syntax

C++

```
template<class RealType = double>
class piecewise_linear_distribution
   {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructor and reset functions
   piecewise_linear_distribution();
   template <class InputIteratorI, class InputIteratorW>
   piecewise_linear_distribution(
      InputIteratorI firstI, InputIteratorI lastI, InputIteratorW firstW);
   template <class UnaryOperation>
   piecewise_linear_distribution(
      initializer_list<result_type> intervals, UnaryOperation weightfunc);
   template <class UnaryOperation>
   piecewise_linear_distribution(
      size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
   explicit piecewise_linear_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   vector<result_type> intervals() const;
   vector<result_type> densities() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating point result type, defaults to `double`. For possible types, see <random>.

Remarks

This sampling distribution has varying-width intervals with linearly varying probability in each interval. For information about other sampling distributions, see piecewise_linear_distribution and discrete_distribution.

The following table links to articles about individual members:

piecewise_linear_distribution
param_type

The property function `intervals()` returns a `vector<result_type>` with the set of stored intervals of the distribution.

The property function `densities()` returns a `vector<result_type>` with the stored densities for each interval set, which are calculated according to the weights provided in the constructor parameters.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

using namespace std;

void test(const int s) {

    // uncomment to use a non-deterministic generator
    // random_device rd;
    // mt19937 gen(rd());
    mt19937 gen(1701);

    // Three intervals, non-uniform: 0 to 1, 1 to 6, and 6 to 15
    vector<double> intervals{ 0, 1, 6, 15 };
    // weights determine the densities used by the distribution
```

```
    vector<double> weights{ 1, 5, 5, 10 };

    piecewise_linear_distribution<double> distr(intervals.begin(), intervals.end(),
weights.begin());

    cout << endl;
    cout << "min() == " << distr.min() << endl;
    cout << "max() == " << distr.max() << endl;
    cout << "intervals (index: interval):" << endl;
    vector<double> i = distr.intervals();
    int counter = 0;
    for (const auto& n : i) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) <<
n << endl;
        ++counter;
    }
    cout << endl;
    cout << "densities (index: density):" << endl;
    vector<double> d = distr.densities();
    counter = 0;
    for (const auto& n : d) {
        cout << fixed << setw(11) << counter << ": " << setw(14) << setprecision(10) <<
n << endl;
        ++counter;
    }
    cout << endl;

    // generate the distribution as a histogram
    map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    cout << "Distribution for " << s << " samples:" << endl;
    for (const auto& elem : histogram) {
        cout << setw(5) << elem.first << '-' << elem.first + 1 << ' ' <<
string(elem.second, ':') << endl;
    }
    cout << endl;
}

int main()
{
    int samples = 100;

    cout << "Use CTRL-Z to bypass data entry and run using default values." << endl;
    cout << "Enter an integer value for the sample count: ";
    cin >> samples;

    test(samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for the sample count: 100
min() == 0
max() == 15
intervals (index: interval):
```

```
          0:   0.0000000000
          1:   1.0000000000
          2:   6.0000000000
          3:  15.0000000000
densities (index: density):
          0:   0.0645161290
          1:   0.3225806452
          2:   0.3225806452
          3:   0.6451612903
Distribution for 100 samples:
    0-1 ::::::::::::::::::::
    1-2 ::::::
    2-3 :::
    3-4 :::::::
    4-5 ::::::
    5-6 ::::::
    6-7 :::::
    7-8 ::::::::::
    8-9 ::::::::::
    9-10 ::::::
   10-11 ::::
   11-12 :::
   12-13 :::
   13-14 :::::
   14-15 :::::
```

Requirements

Header: <random>

Namespace: std

piecewise_linear_distribution::piecewise_linear_distribution

Constructs the distribution.

C++

```
// default constructor
piecewise_linear_distribution();

// constructs using a range of intervals, [firstI, lastI), with
// matching weights starting at firstW
template <class InputIteratorI, class InputIteratorW>
piecewise_linear_distribution(InputIteratorI firstI, InputIteratorI lastI,
InputIteratorW firstW);

// constructs using an initializer list for range of intervals,
// with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_linear_distribution(initializer_list<RealType>
intervals, UnaryOperation weightfunc);

// constructs using an initializer list for range of count intervals,
// distributed uniformly over [xmin,xmax] with weights generated by function weightfunc
template <class UnaryOperation>
piecewise_linear_distribution(size_t count, RealType xmin, RealType xmax, UnaryOperation
```

```
weightfunc);

// constructs from an existing param_type structure
explicit piecewise_linear_distribution(const param_type& parm);
```

Parameters

firstI
An input iterator of the first element in the distribution range.

lastI
An input iterator of the last element in the distribution range.

firstW
An input iterator of the first element in the weights range.

intervals
An initializer_list with the intervals of the distribution.

count
The number of elements in the distribution range.

xmin
The lowest value in the distribution range.

xmax
The highest value in the distribution range. Must be greater than *xmin*.

weightfunc
The object representing the probability function for the distribution. Both the parameter and the return value must be convertible to `double`.

parm
The parameter structure used to construct the distribution.

Remarks

The default constructor sets the stored parameters such that there is one interval, 0 to 1, with a probability density of 1.

The iterator range constructor

C++

```
template <class InputIteratorI, class InputIteratorW>
piecewise_linear_distribution(
    InputIteratorI firstI,
    InputIteratorI lastI,
    InputIteratorW firstW);
```

constructs a distribution object with itnervals from iterators over the sequence [`firstI`, `lastI`) and a matching weight sequence starting at *firstW*.

The initializer list constructor

```
C++
```

```
template <class UnaryOperation>
piecewise_linear_distribution(
    initializer_list<result_type> intervals,
    UnaryOperation weightfunc);
```

constructs a distribution object with intervals from the initializer list *intervals* and weights generated from the function *weightfunc*.

The constructor defined as

```
C++
```

```
template <class UnaryOperation>
piecewise_linear_distribution(
    size_t count,
    result_type xmin,
    result_type xmax,
    UnaryOperation weightfunc);
```

constructs a distribution object with *count* intervals distributed uniformly over [`xmin,xmax`], assigning each interval weights according to the function *weightfunc*, and *weightfunc* must accept one parameter and have a return value, both of which are convertible to `double`. **Precondition:** `xmin < xmax`.

The constructor defined as

```
C++
```

```
explicit piecewise_linear_distribution(const param_type& parm);
```

constructs a distribution object using *parm* as the stored parameter structure.

piecewise_linear_distribution::param_type

Stores all the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef piecewise_linear_distribution<result_type> distribution_type;
   param_type();
   template <class IterI, class IterW>
   param_type(
      IterI firstI, IterI lastI, IterW firstW);
   template <class UnaryOperation>
```

```
    param_type(
       size_t count, result_type xmin, result_type xmax, UnaryOperation weightfunc);
    std::vector<result_type> densities() const;
    std::vector<result_type> intervals() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
    };
```

Parameters

See constructor parameters for piecewise_linear_distribution.

Remarks

Precondition: `xmin < xmax`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

poisson_distribution Class

Article • 08/03/2021

Generates a Poisson distribution.

Syntax

C++

```
template<class IntType = int>
class poisson_distribution
   {
public:
   // types
   typedef IntType result_type;
   struct param_type;

   // constructors and reset functions
   explicit poisson_distribution(double mean = 1.0);
   explicit poisson_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   double mean() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

IntType
The integer result type, defaults to `int`. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type with a Poisson distribution. The following table links to articles about

individual members.

poisson_distribution
param_type

The property function `mean()` returns the value for stored distribution parameter *mean*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the Poisson distribution, see the Wolfram MathWorld article Poisson Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double p, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::poisson_distribution<> distr(p);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "p() == " << std::fixed << std::setw(11) <<
```

```
std::setprecision(10) << distr.mean() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' <<
std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double p_dist = 1.0;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'mean' distribution
parameter (must be greater than zero): ";
    std::cin >> p_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(p_dist, samples);
}
```

First test:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 100
min() == 0
max() == 2147483647
p() == 1.0000000000
Distribution for 100 samples:
    0 ::::::::::::::::::::::::::::
    1 ::::::::::::::::::::::::::::::::::::
    2 ::::::::::::::::::::::
    3 ::::::::
    5 :
```

Second test:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'mean' distribution parameter (must be
greater than zero): 10
Enter an integer value for the sample count: 100
min() == 0
max() == 2147483647
p() == 10.0000000000
Distribution for 100 samples:
    3 :
    4 ::
    5 ::
    6 ::::::::
    7 ::::
    8 ::::::::
    9 ::::::::::::::
   10 ::::::::::::
   11 ::::::::::::::::
   12 ::::::::::::::
   13 ::::::::
   14 ::::::
   15 :
   16 ::
   17 :
```

Requirements

Header: <random>

Namespace: std

poisson_distribution::poisson_distribution

Constructs the distribution.

C++

```
explicit poisson_distribution(RealType mean = 1.0);
explicit binomial_distribution(const param_type& parm);
```

Parameters

mean
The `mean` distribution parameter.

parm
The parameter structure used to construct the distribution.

Remarks

Precondition: `0.0 < mean`

The first constructor constructs an object whose stored `mean` value holds the value *mean*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

poisson_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef poisson_distribution<IntType> distribution_type;
   param_type(double mean = 1.0);
   double mean() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

See constructor parameters for poisson_distribution.

Remarks

Precondition: `0.0 < mean`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

random_device Class

Article • 08/03/2021

Generates a random sequence from an external device.

Syntax

C++

```
class random_device {
public:
   typedef unsigned int result_type;

   // constructor
   explicit random_device(const std::string& token = "");

   // properties
   static result_type min();
   static result_type max();
   double entropy() const;

   // generate
   result_type operator()();

   // no-copy functions
   random_device(const random_device&) = delete;
   void operator=(const random_device&) = delete;
   };
```

Members

random_device

entropy

random_device::operator()

Remarks

The class describes a source of random numbers, and is allowed but not required to be non-deterministic or cryptographically secure by the ISO C++ Standard. In the Visual Studio implementation the values produced are non-deterministic and cryptographically secure, but runs more slowly than generators created from engines and engine adaptors (such as mersenne_twister_engine, the high quality and fast engine of choice for most applications).

`random_device` results are uniformly distributed in the closed range [`0, 2`^{32}).

`random_device` is not guaranteed to result in a non-blocking call.

Generally, `random_device` is used to seed other generators created with engines or engine adaptors. For more information, see <random>.

Example

The following code demonstrates basic functionality of this class and example results. Because of the non-deterministic nature of `random_device`, the random values shown in the **Output** section will not match your results. This is normal and expected.

C++

```
// random_device_engine.cpp
// cl.exe /W4 /nologo /EHsc /MTd
#include <random>
#include <iostream>
using namespace std;

int main()
{
    random_device gen;

    cout << "entropy == " << gen.entropy() << endl;
    cout << "min == " << gen.min() << endl;
    cout << "max == " << gen.max() << endl;

    cout << "a random value == " << gen() << endl;
    cout << "a random value == " << gen() << endl;
    cout << "a random value == " << gen() << endl;
}
```

Output

```
entropy == 32
min == 0
max == 4294967295
a random value == 2378414971
a random value == 3633694716
a random value == 213725214
```

This example is simplistic and not representative of the general use-case for this generator. For a more representative code example, see <random>.

Requirements

Header: <random>

Namespace: std

random_device::random_device

Constructs the generator.

```
C++
```

```
random_device(const std::string& = "");
```

Remarks

The constructor initializes the generator as needed, ignoring the string parameter. Throws a value of an implementation-defined type derived from exception if the `random_device` could not be initialized.

random_device::entropy

Estimates the randomness of the source.

```
C++
```

```
double entropy() const noexcept;
```

Remarks

The member function returns an estimate of the randomness of the source, as measured in bits.

random_device::operator()

Returns a random value.

```
C++
```

```
result_type operator()();
```

Remarks

Returns values uniformly distributed in the closed interval [`min, max`] as determined by member functions `min()` and `max()`. Throws a value of an implementation-defined type derived from exception if a random number could not be obtained.

See also

<random>

seed_seq Class

Article • 08/03/2021

Stores a vector of unsigned integer values that can supply a randomized seed for a random-number engine.

Syntax

C++

```
class seed_seq
   {
public:
   // types
   typedef unsigned int result_type;

   // constructors
   seed_seq();
   template <class T>
      seed_seq(initializer_list<T> initlist);
   template <class InputIterator>
      seed_seq(InputIterator begin, InputIterator end);

   // generating functions
   template <class RandomAccessIterator>
      void generate(RandomAccessIterator begin, RandomAccessIterator end);

   // property functions
   size_t size() const;
   template <class OutputIterator>
      void param(OutputIterator dest) const;

   // no copy functions
   seed_seq(const seed_seq&) = delete;
   void operator=(const seed_seq&) = delete;
   };
```

Types

C++

```
typedef unsigned int result_type;
```

The type of the elements of the seed sequence. A 32-bit unsigned integer type.

Constructors

```
C++

seed_seq();
```

Default constructor, initializes to have an empty internal sequence.

```
C++

template<class T>
seed_seq(initializer_list<T> initlist);
```

Uses `initlist` to set the internal sequence. `T` must be an integer type.

```
C++

template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);
```

Initializes the internal sequence using all elements in the input iterator range provided. `iterator_traits<InputIterator>::value_type` must be an integer type.

Members

Generating Functions

```
C++

template<class RandomAccessIterator>
void generate(RandomAccessIterator begin,
          RandomAccessIterator end);
```

Populates the elements of the provided sequence using an internal algorithm. This algorithm is affected by the internal sequence with which `seed_seq` was initialized. Does nothing if `begin == end`.

Property Functions

```
C++
```

```
size_t size() const;
```

Returns the number of elements in the `seed_seq`.

C++

```
template<class OutputIterator>
void param(OutputIterator dest) const;
```

Copies the internal sequence into the output iterator `dest`.

Example

The following code example exercises the three constructors and generates output from the resulting `seed_seq` instances when assigned to an array. For an example that uses `seed_seq` with a random number generator, see <random>.

C++

```
#include <iostream>
#include <random>
#include <string>
#include <array>

using namespace std;

void test(const seed_seq& sseq) {
    cout << endl << "seed_seq::size(): " << sseq.size() << endl;

    cout << "seed_seq::param(): ";
    ostream_iterator<unsigned int> out(cout, " ");
    sseq.param(out);
    cout << endl;

    cout << "Generating a sequence of 5 elements into an array: " << endl;
    array<unsigned int, 5> seq;
    sseq.generate(seq.begin(), seq.end());
    for (unsigned x : seq) { cout << x << endl; }
}

int main()
{
    seed_seq seed1;
    test(seed1);

    seed_seq seed2 = { 1701, 1729, 1791 };
    test(seed2);
```

```
    string sstr = "A B C D"; // seed string
    seed_seq seed3(sstr.begin(), sstr.end());
    test(seed3);
}
```

Output

```
seed_seq::size(): 0
seed_seq::param():
Generating a sequence of 5 elements into an array:
505382999
163489202
3932644188
763126080
73937346

seed_seq::size(): 3
seed_seq::param(): 1701 1729 1791
Generating a sequence of 5 elements into an array:
1730669648
1954224479
2809786021
1172893117
2393473414

seed_seq::size(): 7
seed_seq::param(): 65 32 66 32 67 32 68
Generating a sequence of 5 elements into an array:
3139879222
3775111734
1084804564
2485037668
1985355432
```

Remarks

Member functions of this class do not throw exceptions.

Requirements

Header: <random>

Namespace: std

See also

<random>

shuffle_order_engine Class

Article • 08/03/2021

Generates a random sequence by reordering the values returned from its base engine.

Syntax

```
C++
```

```
template <class Engine, size_t K>
class shuffle_order_engine;
```

Parameters

Engine
The base engine type.

K
Table size. Number of elements in the buffer (table). **Precondition**: `0 < K`

Members

`shuffle_order_engine::shuffle_order_engine`

`shuffle_order_engine::base`

`shuffle_order_engine::base_type`

`shuffle_order_engine::discard`

`shuffle_order_engine::operator()`

`shuffle_order_engine::seed`

For more information about engine members, see <random>.

Remarks

This class template describes an *engine adaptor* that produces values by reordering the values returned by its base engine. Each constructor fills the internal table with *K* values returned by the base engine, and a random element is selected from the table when a value is requested.

Requirements

Header: <random>

Namespace: std

See also

<random>

student_t_distribution Class

Article • 08/03/2021

Generates a Student's *t*-distribution.

Syntax

C++

```
template<class RealType = double>
class student_t_distribution {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructor and reset functions
   explicit student_t_distribution(result_type n = 1.0);
   explicit student_t_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
   result_type operator()(URNG& gen);
   template <class URNG>
   result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type n() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType

The floating-point result type, defaults to `double`. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified integral type, or type `double` if none is provided, distributed according to the Student's

t-Distribution. The following table links to articles about individual members.

student_t_distribution
param_type

The property function `n()` returns the value for the stored distribution parameter `n`.

For more information about distribution classes and their members, see <random>.

For detailed information about the Student's *t*-distribution, see the Wolfram MathWorld article Students t-Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double n, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::student_t_distribution<> distr(n);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "n() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.n() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
```

```
    std::cout << std::endl;
}

int main()
{
    double n_dist = 0.5;
    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'n' distribution
parameter (must be greater than zero): ";
    std::cin >> n_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(n_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'n' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == -1.79769e+308
max() == 1.79769e+308
n() == 1.0000000000
Distribution for 10 samples:
    1: -1.3084956212
    2: -1.0899518684
    3: -0.9568771388
    4: -0.9372088821
    5: -0.7381334669
    6: -0.2488074854
    7: -0.2028714601
    8: 1.4013074495
    9: 5.3244792236
    10: 92.7084335614
```

Requirements

Header: <random>

Namespace: std

student_t_distribution::student_t_distribution

Constructs the distribution.

```
C++
```

```
explicit student_t_distribution(RealType n = 1.0);
explicit student_t_distribution(const param_type& parm);
```

Parameters

n
The `n` distribution parameter.

parm
The parameter package used to construct the distribution.

Remarks

Precondition: `0.0 < n`

The first constructor constructs an object whose stored `n` value holds the value *n*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

student_t_distribution::param_type

Stores all the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef student_t_distribution<result_type> distribution_type;
   param_type(result_type n = 1.0);
   result_type n() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

n
The `n` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < n`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

subtract_with_carry_engine Class

Article • 08/03/2021

Generates a random sequence by the subtract-with-carry (lagged Fibonacci) algorithm.

Syntax

```
C++
```

```
template <class UIntType, size_t W, size_t S, size_t R>
class subtract_with_carry_engine;
```

Parameters

UIntType
The unsigned integer result type. For possible types, see <random>.

W
Word size. Size of each word, in bits, of the state sequence. **Precondition**: `0 < W ≤ numeric_limits<UIntType>::digits`

S
Short lag. Number of integer values. **Precondition**: `0 < S < R`

R
Long lag. Determines recurrence in the series generated.

Members

`subtract_with_carry_engine::subtract_with_carry_engine`

`subtract_with_carry_engine::max`

`subtract_with_carry_engine::min`

`subtract_with_carry_engine::discard`

`subtract_with_carry_engine::operator()`

`subtract_with_carry_engine::seed`

`default_seed` is a member constant, defined as `19780503u`, used as the default parameter value for `subtract_with_carry_engine::seed` and the single value constructor.

For more information about engine members, see <random>.

Remarks

The `substract_with_carry_engine` class template is an improvement over the linear_congruential_engine. Neither for these engines is as fast or with as high quality results as the mersenne_twister_engine.

This engine produces values of a user-specified unsigned integral type using the recurrence relation (*period*) `x(i) = (x(i - R) - x(i - S) - cy(i - 1)) mod M`, where `cy(i)` has the value `1` if `x(i - S) - x(i - R) - cy(i - 1) < 0`, otherwise `0`, and `M` has the value 2^W. The engine's state is a carry indicator plus *R* values. These values consist of the last *R* values returned if `operator()` has been called at least *R* times, otherwise the `N` values that have been returned and the last `R - N` values of the seed.

The template argument `UIntType` must be large enough to hold values up to `M - 1`.

Although you can construct a generator from this engine directly, you can also use one of these predefined typedefs:

`ranlux24_base`: Used as a base for `ranlux24`. `typedef subtract_with_carry_engine<unsigned int, 24, 10, 24> ranlux24_base;`

`ranlux48_base`: Used as a base for `ranlux48`. `typedef subtract_with_carry_engine<unsigned long long, 48, 5, 12> ranlux48_base;`

For detailed information about the subract with carry engine algorithm, see the Wikipedia article Lagged Fibonacci generator.

Requirements

Header: <random>

Namespace: std

See also

<random>

uniform_int_distribution Class

Article • 08/03/2021

Generates a uniform (every value is equally probable) integer distribution within an output range that is inclusive-inclusive.

Syntax

C++

```
template<class IntType = int>
   class uniform_int_distribution {
public:
   // types
   typedef IntType result_type;
   struct param_type;

   // constructors and reset functions
   explicit uniform_int_distribution(
      result_type a = 0, result_type b =
numeric_limits<result_type>::max());
   explicit uniform_int_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
      result_type operator()(URNG& gen);
   template <class URNG>
      result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type a() const;
   result_type b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
};
```

Parameters

IntType
The integer result type, defaults to `int`. For possible types, see <random>.

Remarks

The class template describes an inclusive-inclusive distribution that produces values of a user-specified integral type with a distribution so that every value is equally probable. The following table links to articles about individual members.

uniform_int_distribution
param_type

The property member `a()` returns the currently stored minimum bound of the distribution, while `b()` returns the currently stored maximum bound. For this distribution class, these minimum and maximum values are the same as those returned by the common property functions `min()` and `max()`.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const int a, const int b, const int s) {

    // uncomment to use a non-deterministic seed
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::uniform_int_distribution<> distr(a, b);
```

```
    std::cout << "lower bound == " << distr.a() << std::endl;
    std::cout << "upper bound == " << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<int, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    for (const auto& elem : histogram) {
        std::cout << std::setw(5) << elem.first << ' ' <<
std::string(elem.second, ':') << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    int a_dist = 1;
    int b_dist = 10;

    int samples = 100;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter an integer value for the lower bound of the
distribution: ";
    std::cin >> a_dist;
    std::cout << "Enter an integer value for the upper bound of the
distribution: ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter an integer value for the lower bound of the distribution: 0
Enter an integer value for the upper bound of the distribution: 12
Enter an integer value for the sample count: 200
lower bound == 0
upper bound == 12
Distribution for 200 samples:
    0 ::::::::::::::
    1 ::::::::::::::::::::
    2 ::::::::::::::::::
    3 ::::::::::::::
```

```
 4 :::::::
 5 ::::::::::::::::::::
 6 ::::::::::::
 7 ::::::::::
 8 ::::::::::::::
 9 ::::::::::::
10 ::::::::::::::::::::
11 ::::::::::::
12 ::::::::::::::::
```

Requirements

Header: <random>

Namespace: std

uniform_int_distribution::uniform_int_distribution

Constructs the distribution.

C++

```
explicit uniform_int_distribution(
   result_type a = 0, result_type b =
std::numeric_limits<result_type>::max());
explicit uniform_int_distribution(const param_type& parm);
```

Parameters

a

The lower bound for random values, inclusive.

b

The upper bound for random values, inclusive.

parm

The `param_type` structure used to construct the distribution.

Remarks

Precondition: `a ≤ b`

The first constructor constructs an object whose stored *a* value holds the value *a* and whose stored *b* value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

uniform_int_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef uniform_int_distribution<result_type> distribution_type;
   param_type(
      result_type a = 0, result_type b =
std::numeric_limits<result_type>::max());
   result_type a() const;
   result_type b() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

a
The lower bound for random values, inclusive.

b
The upper bound for random values, inclusive.

right
The `param_type` object to compare to this.

Remarks

Precondition: `a ≤ b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

uniform_real_distribution Class

Article • 08/03/2021

Generates a uniform (every value is equally probable) floating-point distribution within an output range that is inclusive-exclusive.

Syntax

C++

```
template<class RealType = double>
   class uniform_real_distribution {
public:
   // types
   typedef RealType result_type;
   struct param_type;

   // constructors and reset functions
   explicit uniform_real_distribution(
      result_type a = 0.0, result_type b = 1.0);
   explicit uniform_real_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
      result_type operator()(URNG& gen);
   template <class URNG>
      result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type a() const;
   result_type b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
};
```

Parameters

RealType
The floating-point result type, defaults to **`double`**. For possible types, see <random>.

Remarks

The class template describes an inclusive-exclusive distribution that produces values of a user-specified integral floating point type with a distribution so that every value is equally probable. The following table links to articles about individual members.

uniform_real_distribution
param_type

The property member `a()` returns the currently stored minimum bound of the distribution, while `b()` returns the currently stored maximum bound. For this distribution class, these minimum and maximum values are the same as those returned by the common property functions `min()` and `max()` described in the <random> topic.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic seed
    //    std::random_device rd;
    //    std::mt19937 gen(rd());
    std::mt19937 gen(1729);

    std::uniform_real_distribution<> distr(a,b);
```

```
    std::cout << "lower bound == " << distr.a() << std::endl;
    std::cout << "upper bound == " << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setprecision(10) << elem.first << std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 1.0;
    double b_dist = 1.5;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the lower bound of the
distribution: ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the upper bound of the
distribution: ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the lower bound of the distribution: 0
Enter a floating point value for the upper bound of the distribution: 1
Enter an integer value for the sample count: 10
lower bound == 0
upper bound == 1
Distribution for 10 samples:
          1: 0.0288609485
          2: 0.2007994386
          3: 0.3027480117
```

```
 4: 0.4124758695
 5: 0.4413777133
 6: 0.4846447405
 7: 0.6225745916
 8: 0.6880935217
 9: 0.7541936723
10: 0.8795716566
```

Requirements

Header: <random>

Namespace: std

uniform_real_distribution::uniform_real_distribution

Constructs the distribution.

```
C++
```

```
explicit uniform_real_distribution(result_type a = 0.0, result_type b =
1.0);
explicit uniform_real_distribution(const param_type& parm);
```

Parameters

a
The lower bound for random values, inclusive.

b
The upper bound for random values, exclusive.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `a < b`

The first constructor constructs an object whose stored *a* value holds the value *a* and whose stored *b* value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

uniform_real_distribution::param_type

Stores all the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef uniform_real_distribution<result_type> distribution_type;
   param_type(result_type a = 0.0, result_type b = 1.0);
   result_type a() const;
   result_type b() const;

   bool operator==(const param_type& right) const;
   bool operator!=(const param_type& right) const;
   };
```

Parameters

a
The lower bound for random values, inclusive.

b
The upper bound for random values, exclusive.

right
The `param_type` object to compare to this.

Remarks

Precondition: `a < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

weibull_distribution Class

Article • 08/03/2021

Generates a Weibull distribution.

Syntax

C++

```
class weibull_distribution
   {
   public:
    // types
   typedef RealType result_type;
   struct param_type;

    // constructor and reset functions
   explicit weibull_distribution(result_type a = 1.0, result_type b = 1.0);
   explicit weibull_distribution(const param_type& parm);
   void reset();

   // generating functions
   template <class URNG>
      result_type operator()(URNG& gen);
   template <class URNG>
      result_type operator()(URNG& gen, const param_type& parm);

   // property functions
   result_type a() const;
   result_type b() const;
   param_type param() const;
   void param(const param_type& parm);
   result_type min() const;
   result_type max() const;
   };
```

Parameters

RealType
The floating-point result type, defaults to `double`. For possible types, see <random>.

Remarks

The class template describes a distribution that produces values of a user-specified floating point type, or type `double` if none is provided, distributed according to the

Weibull Distribution. The following table links to articles about individual members.

weibull_distribution
param_type

The property functions `a()` and `b()` return their respective values for stored distribution parameters *a* and *b*.

The property member `param()` sets or returns the `param_type` stored distribution parameter package.

The `min()` and `max()` member functions return the smallest possible result and largest possible result, respectively.

The `reset()` member function discards any cached values, so that the result of the next call to `operator()` does not depend on any values obtained from the engine before the call.

The `operator()` member functions return the next generated value based on the URNG engine, either from the current parameter package, or the specified parameter package.

For more information about distribution classes and their members, see <random>.

For detailed information about the Weibull distribution, see the Wolfram MathWorld article Weibull Distribution.

Example

C++

```
// compile with: /EHsc /W4
#include <random>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>

void test(const double a, const double b, const int s) {

    // uncomment to use a non-deterministic generator
    //    std::random_device gen;
    std::mt19937 gen(1701);

    std::weibull_distribution<> distr(a, b);

    std::cout << std::endl;
    std::cout << "min() == " << distr.min() << std::endl;
```

```
    std::cout << "max() == " << distr.max() << std::endl;
    std::cout << "a() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.a() << std::endl;
    std::cout << "b() == " << std::fixed << std::setw(11) <<
std::setprecision(10) << distr.b() << std::endl;

    // generate the distribution as a histogram
    std::map<double, int> histogram;
    for (int i = 0; i < s; ++i) {
        ++histogram[distr(gen)];
    }

    // print results
    std::cout << "Distribution for " << s << " samples:" << std::endl;
    int counter = 0;
    for (const auto& elem : histogram) {
        std::cout << std::fixed << std::setw(11) << ++counter << ": "
            << std::setw(14) << std::setprecision(10) << elem.first <<
std::endl;
    }
    std::cout << std::endl;
}

int main()
{
    double a_dist = 0.0;
    double b_dist = 1;

    int samples = 10;

    std::cout << "Use CTRL-Z to bypass data entry and run using default
values." << std::endl;
    std::cout << "Enter a floating point value for the 'a' distribution
parameter (must be greater than zero): ";
    std::cin >> a_dist;
    std::cout << "Enter a floating point value for the 'b' distribution
parameter (must be greater than zero): ";
    std::cin >> b_dist;
    std::cout << "Enter an integer value for the sample count: ";
    std::cin >> samples;

    test(a_dist, b_dist, samples);
}
```

Output

First run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter (must be
```

```
greater than zero): 1
Enter a floating point value for the 'b' distribution parameter (must be
greater than zero): 1
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
a() == 1.0000000000
b() == 1.0000000000
Distribution for 10 samples:
    1: 0.0936880533
    2: 0.1225944894
    3: 0.6443593183
    4: 0.6551171649
    5: 0.7313457551
    6: 0.7313557977
    7: 0.7590097389
    8: 1.4466885214
    9: 1.6434088411
    10: 2.1201210996
```

Second run:

Output

```
Use CTRL-Z to bypass data entry and run using default values.
Enter a floating point value for the 'a' distribution parameter (must be
greater than zero): .5
Enter a floating point value for the 'b' distribution parameter (must be
greater than zero): 5.5
Enter an integer value for the sample count: 10

min() == 0
max() == 1.79769e+308
a() == 0.5000000000
b() == 5.5000000000
Distribution for 10 samples:
    1: 0.0482759823
    2: 0.0826617486
    3: 2.2835941207
    4: 2.3604817485
    5: 2.9417663742
    6: 2.9418471657
    7: 3.1685268104
    8: 11.5109922290
    9: 14.8543594043
    10: 24.7220241239
```

Requirements

Header: <random>

Namespace: std

weibull_distribution::weibull_distribution

```
C++
```

```
explicit weibull_distribution(result_type a = 1.0, result_type b = 1.0);
explicit weibull_distribution(const param_type& parm);
```

Parameters

a
The `a` distribution parameter.

b
The `b` distribution parameter.

parm
The `param_type` structure used to construct the distribution.

Remarks

Precondition: `0.0 < a` and `0.0 < b`

The first constructor constructs an object whose stored `a` value holds the value *a* and whose stored `b` value holds the value *b*.

The second constructor constructs an object whose stored parameters are initialized from *parm*. You can obtain and set the current parameters of an existing distribution by calling the `param()` member function.

weibull_distribution::param_type

Stores the parameters of the distribution.

```
C++
```

```
struct param_type {
   typedef weibull_distribution<result_type> distribution_type;
   param_type(result_type a = 1.0, result_type b = 1.0);
```

```
    result_type a() const;
    result_type b() const;

    bool operator==(const param_type& right) const;
    bool operator!=(const param_type& right) const;
    };
```

Parameters

a
The `a` distribution parameter.

b
The `b` distribution parameter.

right
The `param_type` object to compare to this.

Remarks

Precondition: `0.0 < a` and `0.0 < b`

This structure can be passed to the distribution's class constructor at instantiation, to the `param()` member function to set the stored parameters of an existing distribution, and to `operator()` to be used in place of the stored parameters.

See also

<random>

<ranges>

Article • 01/30/2023

At a high level, a *range* is something that you can iterate over. A range is represented by an iterator that marks the beginning of the range and a sentinel that marks the end of the range. The sentinel may be the same type as the begin iterator, or it may be different. The containers, such as `vector` and `list`, in the C++ Standard Library are ranges. A range abstracts iterators in a way that simplifies and amplifies your ability to use the Standard Template Library (STL).

STL algorithms usually take iterators that point to the portion of the collection that they should operate on. For example, consider how you sort a `vector` by using `std::sort()`. You pass two iterators that mark the beginning and end of the `vector`. That provides flexibility, but passing the iterators to the algorithm is extra work because you probably just want to sort the whole thing.

With ranges, you can call `std::ranges::sort(myVector);`, which is treated as if you called `std::sort(myVector.begin(), myVector.end());`. In range libraries, algorithms take ranges as parameters (although they can also take iterators, if you want). They can operate directly on collections. Examples of range algorithms available in `<algorithm>` include `copy`, `copy_n`, `copy_if`, `all_of`, `any_of`, `none_of`, `find`, `find_if`, `find_if_not`, `count`, `count_if`, `for_each`, `for_each_n`, `equal`, and `mismatch`.

But perhaps the most important benefit of ranges is that you can compose STL algorithms that operate on ranges in a style that's reminiscent of functional programming.

A ranges example

Before ranges, if you wanted to transform the elements of a collection that met a certain criterion, you needed to introduce an intermediate step to hold the results between operations. For example, if you wanted to build a vector of squares from the elements in another vector that are divisible by three, you could write something like:

C++

```
std::vector<int> input = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
std::vector<int> intermediate, output;

std::copy_if(input.begin(), input.end(), std::back_inserter(intermediate),
[](const int i) { return i%3 == 0; });
```

```
std::transform(intermediate.begin(), intermediate.end(),
std::back_inserter(output), [](const int i) {return i*i; });
```

With ranges, you can accomplish the same thing without needing the `intermediate` vector:

C++

```
// requires /std:c++20
std::vector<int> input = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto output = input
    | std::views::filter([](const int n) {return n % 3 == 0; })
    | std::views::transform([](const int n) {return n * n; });
```

Besides being easier to read, this code avoids the memory allocation that's required for the `intermediate` vector and its contents. It also allows you to compose two operations.

In the preceding code, each element that's divisible by three is combined with an operation to square that element. The pipe (`|`) symbol chains the operations together and is read left to right.

The result, `output`, is itself a kind of range called a *view*.

Views

A view is a lightweight range. View operations--such as default construction, move construction/assignment, copy construction/assignment (if present), destruction, begin, and end--all happen in constant time regardless of the number of elements in the view.

Views are created by range adaptors, which are discussed in the following section. For more information about the classes that implement various views, see View classes.

How the elements in the view appear depends on the range adaptor that you use to create the view. In the previous example, a range adaptor takes a range and returns a view of the elements divisible by three. The underlying range is unchanged.

Views are composable, which is powerful. In the previous example, the view of vector elements that are divisible by three is combined with the view that squares those elements.

The elements of a view are evaluated lazily. That is, the transformations that you apply to each element in a view aren't evaluated until you ask for the element. For example, if you run the following code in a debugger and put a breakpoint on the lines `auto`

`divisible_by_three = ...` and `auto square = ...`, you'll see that you hit the `divisible_by_three` lambda breakpoint as each element in `input` is tested for divisibility by three. The `square` lambda breakpoint will be hit as the elements that are divisible by three are squared.

C++

```
// requires /std:c++20
#include <ranges>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> input =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    auto divisible_by_three = [](const int n) {return n % 3 == 0; };
    auto square = [](const int n) {return n * n; };

    auto x = input | std::views::filter(divisible_by_three)
                   | std::views::transform(square);

    for (int i : x)
    {
        std::cout << i << '\n';
    }
    return 0;
}
```

For more information about views, see <ranges> view classes.

Range adaptors

Range adaptors take a range and produce a view. Range adaptors produce lazily evaluated views. That is, you don't incur the cost of transforming every element in the range to produce the view. You only pay the cost to process an element in the view when you access that element.

In the previous example, the `filter` range adaptor creates a view named `input` that contains the elements that are divisible by three. The `transform` range adaptor takes the view of elements divisible by three and creates a view of those elements squared.

Range adaptors can be chained together (composed), which is the heart of the power and flexibility of ranges. Composing range adaptors allows you to overcome the problem that the previous STL algorithms aren't easily composable.

For more information about creating views, see Range adaptors.

Range algorithms

Some range algorithms take a range argument. An example is `std::ranges::sort(myVector);`.

The range algorithms are almost identical to the corresponding iterator-pair algorithms in the `std` namespace. The difference is that they have concept-enforced constraints, and they accept either range arguments or more iterator-sentinel argument pairs. They can work directly on a container and can be easily chained together.

`<ranges>` functions

The following functions are used to create iterators and sentinels for ranges, and to get the size of a range.

Function	Description
begin C++20	Get an iterator to the first element in the range.
cbegin C++20	Get a `const` iterator to the first element in the range.
cend C++20	Get the sentinel at the end of the `const`-qualified range.
cdata C++20	Get a `const` pointer to the first element in the contiguous range.
crbegin C++20	Get a reverse `const` iterator to the beginning of the range.
crend C++20	Get the sentinel at the end of what `crbegin()` returns.
data C++20	Get a pointer to the first element in the contiguous range.
empty C++20	Determine if the range is empty.
end C++20	Get the sentinel at the end of the range.
rbegin C++20	Get a reverse iterator to the beginning of the range.
rend C++20	Get a reverse iterator to the sentinel at the end of the range.
size C++20	Get the size of the range as an unsigned value.
ssize C++20	Get the size of the range as a signed value.

For more information, see <ranges> functions.

Range concepts

How you iterate over the elements of a range depends on its underlying iterator type. Ranges use C++ concepts that specify which iterator they support.

In C++20, to say that concept *X* refines concept *Y* means that everything that satisfies concept *Y* also satisfies concept *X*. For example: *car*, *bus*, and *truck* all refine *vehicle*.

Some range concepts mirror the hierarchy of iterator categories. The following table lists the range concepts, along with the types of containers that they can be applied to.

Range concept	Description	Supported containers
std::ranges::output_range	Can iterate forward.	
std::ranges::input_range	Can iterate from beginning to end at least once.	`std::forward_list` `std::unordered_map` `std::unordered_multimap` `std::unordered_set` `std::unordered_multiset` `basic_istream_view`
std::ranges::forward_range	Can iterate from beginning to end more than once.	`std::forward_list` `std::unordered_map` `std::unordered_multimap` `std::unordered_set` `std::unordered_multiset`
std::ranges::bidirectional_range	Can iterate forward and backward more than once.	`std::list` `std::map` `std::multimap` `std::multiset` `std::set`
std::ranges::random_access_range	Can access an arbitrary element (in constant time) by using the `[]` operator.	`std::deque`
std::ranges::contiguous_range	The elements are stored in memory consecutively.	`std::array` `std::string` `std::vector`

See <ranges> concepts for more information about these concepts.

`<ranges>` alias templates

The following alias templates determine the types of iterators and sentinels for a range:

Alias template	Description

Alias template	Description
borrowed_iterator_t C++20	Determine if an iterator returned for a `range` refers to a range whose lifetime has ended.
borrowed_subrange_t C++20	Determine if an iterator returned for a `subrange` refers to a subrange whose lifetime has ended.
dangling C++20	Indicates that the returned iterator of a `range`/`subrange` outlives the lifetime of the `range`/`subrange` it refers to.
iterator_t C++20	Returns the iterator type of the specified range type.
range_difference_t C++20	Returns the difference type of the specified range's iterator type.
range_reference_t C++20	Returns the reference type of the specified range's iterator type.
range_rvalue_reference_t C++20	Returns the rvalue reference type for the specified range's iterator type. In other words, the rvalue reference type of the range's elements.
range_size_t C++20	Returns the type used to report the specified range's size.
range_value_t C++20	Returns the value type of the specified range's iterator type. Or in other words, the type of the elements in the range.
sentinel_t C++20	Returns the sentinel type of the specified range.

For more information about these alias templates, see <ranges> alias templates.

See also

<ranges> functions
<ranges> concepts
Range adaptors
Header files reference

`<ranges>` concepts

Article • 01/30/2023

Concepts are a C++20 language feature that constrain template parameters at compile time. They help prevent incorrect template instantiation, specify template argument requirements in a readable form, and provide more succinct template related compiler errors.

Consider the following example, which defines a concept to prevent instantiating a template with a type that doesn't support division:

C++

```
// requires /std:c++20 or later
#include <iostream>

// Definition of dividable concept which requires
// that arguments a & b of type T support division
template <typename T>
concept dividable = requires (T a, T b)
{
    a / b;
};

// Apply the concept to a template.
// The template will only be instantiated if argument T supports division.
// This prevents the template from being instantiated with types that don't
support division.
// This could have been applied to the parameter of a template function, but
because
// most of the concepts in the <ranges> library are applied to classes, this
form is demonstrated.
template <class T> requires dividable<T>
class DivideEmUp
{
public:
    T Divide(T x, T y)
    {
        return x / y;
    }
};

int main()
{
    DivideEmUp<int> dividerOfInts;
    std::cout << dividerOfInts.Divide(6, 3); // outputs 2
    // The following line will not compile because the template can't be
instantiated
    // with char* because char* can be divided
    DivideEmUp<char*> dividerOfCharPtrs; // compiler error: cannot deduce
```

```
template arguments
}
```

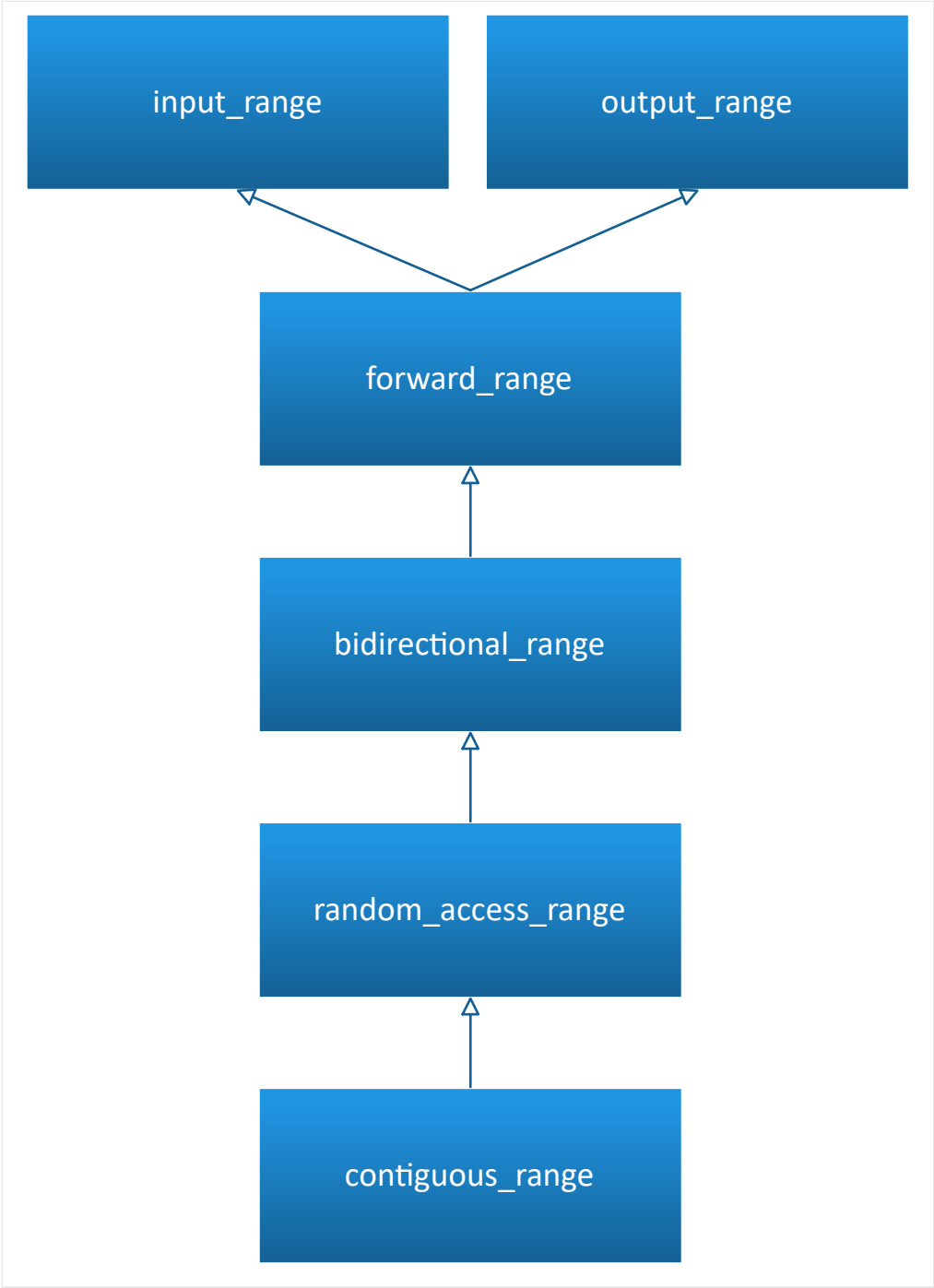
When you pass the compiler switch `/diagnostics:caret` to Visual Studio 2022 version 17.4 preview 4, or later, the error that concept `dividable<char*>` evaluated to false will point directly to the expression requirement `(a / b)` that failed.

Range concepts are defined in the `std::ranges` namespace, and declared in the `<ranges>` header file. They're used in the declarations of range adaptors, views, and so on.

There are six categories of ranges. They're related to the categories of iterators listed in <iterator> concepts. In order of increasing capability, the categories are:

Range concept	Description
output_range input_range	Specifies a range that you can write to. Specifies a range that you can read from once.
forward_range	Specifies a range that you can read (and possibly write) multiple times.
bidirectional_range	Specifies a range that you can read and write both forwards and backwards.
random_access_range	Specifies a range that you can read and write by index.
contiguous_range	Specifies a range whose elements are sequential in memory, are the same size, and can be accessed using pointer arithmetic.

In the preceding table, concepts are listed in order of increasing capability. A range that meets the requirements of a concept generally meets the requirements of the concepts in the rows that precede it. For example, a `random_access_range` has the capability of a `bidirectional_range`, `forward_range`, `input_range`, and `output_range`. The exception is `input_range`, which can't be written to, so it doesn't have the capabilities of `output_range`.



Other range concepts include:

Range concept	Description
range[C++20]	Specifies a type that provides an iterator and a sentinel.
borrowed_range[C++20]	Specifies that the lifetime of the range's iterators aren't tied to the range's lifetime.
common_range[C++20]	Specifies that the type of the range's iterator and the type of the range's sentinel are the same.
Simple_View[C++20]	Not an official concept defined as part of the standard library, but used as a helper concept on some interfaces.

Range concept	Description
sized_range[C++20]	Specifies a range that can provide its number of elements efficiently.
view[C++20]	Specifies a type that has efficient (constant time) move construction, assignment, and destruction.
viewable_range[C++20]	Specifies a type that either is a view or can be converted to one.

`bidirectional_range`

A `bidirectional_range` supports reading and writing the range forwards and backwards.

C++

```
template<class T>
concept bidirectional_range =
    forward_range<T> && bidirectional_iterator<iterator_t<T>>;
```

Parameters

`T`
The type to test to see if it's a `bidirectional_range`.

Remarks

This kind of range supports bidirectional_iterator or greater. A `bidirectional_iterator` has the capabilities of a forward_iterator, but can also iterate backwards.

Some examples of a `bidirectional_range` are `std::set`, `std::vector`, and `std::list`.

`borrowed_range`

A type models `borrowed_range` if the validity of iterators you get from the object can outlive the lifetime of the object. That is, the iterators for a range can be used even when the range no longer exists.

C++

```
template<class T>
concept borrowed_range =
    range<T> &&
    (is_lvalue_reference_v<T> || enable_borrowed_range<remove_cvref_t<T>>);
```

Parameters

`T`
The type to test to see if it's a `borrowed_range`.

Remarks

The lifetime of an rvalue range can end following a function call whether the range models `borrowed_range` or not. If it's a `borrowed_range`, you may be able to continue to use the iterators with well-defined behavior regardless of when the range's lifetime ends.

Cases where this isn't true are, for example, for containers like `vector` or `list` because when the container's lifetime ends, the iterators would refer to elements that have been destroyed.

You can continue to use the iterators for a `borrowed_range`, for example, for a `view` like `iota_view<int>{0, 42}` whose iterators are over set of values that aren't subject to being destroyed because they're generated on demand.

`common_range`

The type of the iterator for a `common_range` is the same as the type of the sentinel. That is, `begin()` and `end()` return the same type.

C++

```
template<class T>
concept common_range =
   ranges::range<T> && std::same_as<ranges::iterator_t<T>,
ranges::sentinel_t<T>>;
```

Parameters

`T`
The type to test to see if it's a `common_range`.

Remarks

Getting the type from `std::ranges::begin()` and `std::ranges::end()` is important for algorithms that calculate the distance between two iterators, and for algorithms that accept ranges denoted by iterator pairs.

The standard containers (for example, `vector`) meet the requirements of `common_range`.

`contiguous_range`

The elements of a `contiguous_range` are stored sequentially in memory and can be accessed using pointer arithmetic. For example, an array is a `contiguous_range`.

C++

```
template<class T>
concept contiguous_range =
    random_access_range<T> && contiguous_iterator<iterator_t<T>> &&
    requires(T& t) {{ ranges::data(t) } ->
same_as<add_pointer_t<range_reference_t<T>>>;};
```

Parameters

T

The type to test to see if it's a `contiguous_range`.

Remarks

A `contiguous_range` can be accessed by pointer arithmetic because the elements are laid out sequentially in memory and are the same size. This kind of range supports contiguous_iterator, which is the most flexible of all the iterators.

Some examples of a `contiguous_range` are `std::array`, `std::vector`, and `std::string`.

Example: `contiguous_range`

The following example shows using pointer arithmetic to access a `contiguous_range`:

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
```

```
int main()
{
    // Show that vector is a contiguous_range
    std::vector<int> v = {0,1,2,3,4,5};
    std::cout << std::boolalpha <<
std::ranges::contiguous_range<decltype(v)> << '\n'; // outputs true

    // Show that pointer arithmetic can be used to access the elements of a
contiguous_range
    auto ptr = v.data();
    ptr += 2;
    std::cout << *ptr << '\n'; // outputs 2
}
```

Output

```
true
2
```

`forward_range`

A `forward_range` supports reading (and possibly writing) the range multiple times.

C++

```
template<class T>
concept forward_range = input_range<T> && forward_iterator<iterator_t<T>>;
```

Parameters

`T`

The type to test to see if it's a `forward_range`.

Remarks

This kind of range supports forward_iterator or greater. A `forward_iterator` can iterate over a range multiple times.

`input_range`

An `input_range` is a range that can be read from once.

C++

```
template<class T>
concept input_range = range<T> && input_iterator<iterator_t<T>>;
```

Parameters

`T`

The type to test to see if it's an `input_range`.

Remarks

When a type meets the requirements of `input_range`:

- The `ranges::begin()` function returns an input_iterator. Calling `begin()` more than once on an `input_range` results in undefined behavior.
- You can dereference an `input_iterator` repeatedly, which yields the same value each time. An `input_range` isn't multi-pass. Incrementing an iterator invalidates any copies.
- It can be used with `ranges::for_each`.
- It supports `input_iterator` or greater.

`output_range`

An `output_range` is a range that you can write to.

C++

```
template<class R, class T>
concept output_range = range<R> && output_iterator<iterator_t<R>, T>;
```

Parameters

`R`

The type of the range.

`T`

The type of the data to write to the range.

Remarks

The meaning of `output_iterator<iterator_t<R>, T>` is that the type provides an iterator that can write values of type `T` to a range of type `R`. In other words, it supports output_iterator or greater.

`random_access_range`

A `random_access_range` can read or write a range by index.

C++

```
template<class T>
concept random_access_range =
bidirectional_range<T> && random_access_iterator<iterator_t<T>>;
```

Parameters

T

The type to test to see if it's a `sized_range`.

Remarks

This kind of range supports random_access_iterator or greater. A `random_access_range` has the capabilities of an `input_range`, `output_range`, `forward_range`, and `bidirectional_range`. A `random_access_range` is sortable.

Some examples of a `random_access_range` are `std::vector`, `std::array`, and `std::deque`.

`range`

Defines the requirements a type must meet to be a `range`. A `range` provides an iterator and a sentinel, so that you can iterate over its elements.

C++

```
template<class T>
concept range = requires(T& rg)
{
  ranges::begin(rg);
  ranges::end(rg);
};
```

Parameters

`T`
The type to test to see if it's a `range`.

Remarks

The requirements of a `range` are:

- It can be iterated using `std::ranges::begin()` and `std::ranges::end()`
- `ranges::begin()` and `ranges::end()` run in amortized constant time and don't modify the `range`. Amortized constant time doesn't mean O(1), but that the average cost over a series of calls, even in the worst case, is O(n) rather than O(n^2) or worse.
- `[ranges::begin(), ranges::end())` denotes a valid range.

`Simple_View`

A `Simple_View` is an exposition-only concept used on some `ranges` interfaces. It isn't defined in the library. It's only used in the specification to help describe the behavior of some range adaptors.

C++

```
template<class V>
  concept Simple_View = // exposition only
    ranges::view<V> && ranges::range<const V> &&
    std::same_as<std::ranges::iterator_t<V>, std::ranges::iterator_t<const
V>> &&
    std::same_as<std::ranges::sentinel_t<V>, std::ranges::sentinel_t<const
V>>;
```

Parameters

`V`
The type to test to see if it's a `Simple_View`.

Remarks

A view `V` is a Simple_View if all of the following are true:

- `V` is a view
- `const V` is a range
- Both `v` and `const V` have the same iterator and sentinel types.

`sized_range`

A `sized_range` provides the number of elements in the range in amortized constant time.

C++

```
template<class T>
  concept sized_range = range<T> &&
    requires(T& t) { ranges::size(t); };
```

Parameters

`T`
The type to test to see if it's a `sized_range`.

Remarks

The requirements of a `sized_range` are that calling `ranges::size` on it:

- Doesn't modify the range.
- Returns the number of elements in amortized constant time. Amortized constant time doesn't mean O(1), but that the average cost over a series of calls, even in the worst case, is O(n) rather than O(n^2) or worse.

Some examples of a `sized_range` are `std::list` and `std::vector`.

Example: `sized_range`

The following example shows that a `vector` of `int` is a `sized_range`:

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
```

```
int main()
{
    std::cout << std::boolalpha <<
std::ranges::sized_range<std::vector<int>> << '\n'; // outputs "true"
}
```

`view`

A `view` has constant time move construction, assignment, and destruction operations--regardless of the number of elements it has. Views don't need to be copy constructible or copy assignable, but if they are, those operations must also run in constant time.

Because of the constant time requirement, you can efficiently compose views. For example, given a vector of `int` called `input`, a function that determines if a number is divisible by three, and a function that squares a number, the statement `auto x = input | std::views::filter(divisible_by_three) | std::views::transform(square);` efficiently produces a view that contains the squares of the numbers in input that are divisible by three. Connecting views together with `|` is referred to as composing the views. If a type satisfies the view concept, then it can be composed efficiently.

C++

```
template<class T>
concept view = ranges::range<T> && std::movable<T> &&
ranges::enable_view<T>;
```

Parameters

T
The type to test to see if it's a view.

Remarks

The essential requirement that makes a view composable is that it's cheap to move/copy. This is because the view is moved/copied when it's composed with another view. It must be a movable range.

`ranges::enable_view<T>` is a trait used to claim conformance to the semantic requirements of the `view` concept. A type can opt in by:

- publicly and unambiguously deriving from a specialization of `ranges::view_interface`

- publicly and unambiguously deriving from the empty class `ranges::view_base`, or
- specializing `ranges::enable_view<T>` to `true`

Option 1 is preferred because `view_interface` also provides default implementation that saves some boilerplate code you have to write.

Failing that, option 2 is a little simpler than option 3.

The advantage of option 3 is that it's possible without changing the definition of the type.

`viewable_range`

A `viewable_range` is a type that either is a view or can be converted to one.

```
C++
```

```
template<class T>
  concept viewable_range =
    range<T> && (borrowed_range<T> || view<remove_cvref_t<T>>);
```

Parameters

T
The type to test to see if it either is a view or can be converted to one.

Remarks

Use `std::ranges::views::all()` to convert a range to a view.

See also

<ranges>
Range adaptors
View classes

`<ranges>` functions

Article • 01/30/2023

The `<ranges>` C++20 header includes the following non-member helper functions.

Non-member functions	Description
begin C++20	Get an iterator to the first element in the range.
cbegin C++20	Get a `const` iterator to the first element in the range.
cend C++20	Get the sentinel at the end of the `const`-qualified range.
cdata C++20	Get a `const` pointer to the first element in the contiguous range.
crbegin C++20	Get a reverse `const` iterator to the beginning of the range.
crend C++20	Get the sentinel at the end of what `crbegin()` returns.
data C++20	Get a pointer to the first element in the contiguous range.
empty C++20	Test if the range is empty.
end C++20	Get the sentinel at the end of the range.
rbegin C++20	Get a reverse iterator to the beginning of the range.
rend C++20	Get a reverse iterator to the sentinel at the end of the range.
size C++20	Get the size of the range as an unsigned value.
ssize C++20	Get the size of the range as a signed value.

Many of these 'functions' are implemented as customization point objects. A customization point object is a function object that can be overloaded on user-defined types, while also enforcing constraints on which kinds of types can be passed to the function object. The net effect is that the compiler figures out if there's a valid customized function to call for the passed in type, or if the default implementation should be used, or if the call is ill-formed.

Many of these functions have corresponding functions in the `std` namespace. But when working with ranges, use these helper functions instead. These functions use C++20 concepts, which provide better compile time errors. Because they're implemented as customization points, problems related to argument dependent lookup (ADL) and const correctness are avoided.

`begin`

Get an iterator to the first element in the range.

```
C++
```

```
template<class T>
constexpr std::input_or_output_iterator auto begin(T&& rg);
```

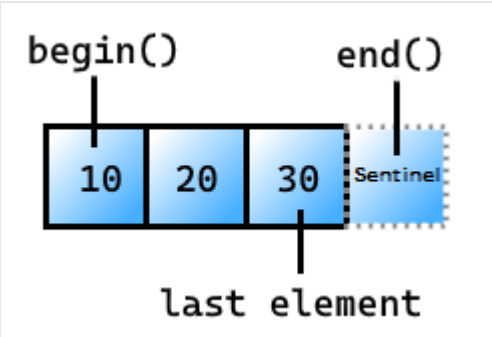
Parameters

`T`
The type of the range.

`rg`
A range.

Return value

An iterator to the first element in the range:



If the range is an array, returns the equivalent of `rg + 0`. If `auto(rg.begin())` yields an iterator, returns the equivalent of `auto(rg.begin())`. If that expression is ill-formed, `auto(begin(rg))` is used if that expression yields an iterator.

Remarks

`ranges::begin()` works on all ranges, whereas `std::begin()` may not.

Example: `begin`

```
C++
```

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> v = {10, 20, 30};
    auto vi = std::ranges::begin(v);
    std::cout << *vi << ' ' << *++vi; // outputs 10 20
}
```

`cbegin`

Get a `const` iterator to the first element in a range. The iterator can access the elements in the range, but can't modify them.

C++

```
template<class T>
constexpr std::input_or_output_iterator auto cbegin(T&& rg);
```

Parameters

`T`

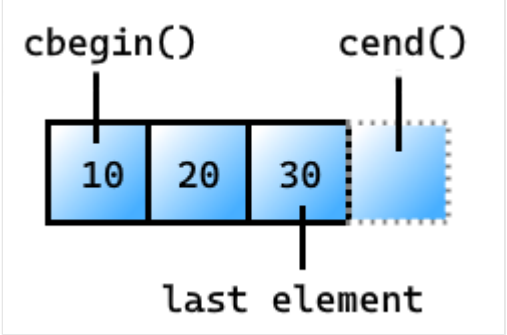
The type of the range.

`rg`

A range.

Return value

A `const` iterator to the first element in the range:



If the range is an array, returns the equivalent of `rg + 0`. If `auto(rg.cbegin())` yields an iterator, returns the equivalent of `auto(rg.cbegin())`. If that expression is ill-formed, `auto(cbegin(rg))` is used if that expression yields an iterator.

Remarks

`ranges::cbegin()` works on all ranges, whereas `std::cbegin()` may not.

Example: `cbegin`

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> v = {10, 20, 30};
    auto vi = std::ranges::cbegin(v);
    std::cout << *vi; // outputs 10
    // *vi = 100; // error because the iterator is const
}
```

`cdata`

Get a `const` pointer to the first element in the contiguous range.

C++

```
template<class T>
constexpr std::add_pointer_t<ranges::range_reference_t<const T>> cdata(T&&
rg);
```

Parameters

`T`

The type of the range.

`rg`

A range.

Return value

A `const` pointer, based on the type of the range, to the first element data in the contiguous range. For example, if the range is a vector of integers, the type of the return value is a `const int *`.

Example: `cdata`

C++

```
#include <ranges>
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    std::string src{ "a string" };

    auto c_charPtr = std::ranges::cdata(src); // ptr is a const char *
    auto c_intPtr = std::ranges::cdata(v); // ptr2 is a const int *
    std::cout << c_charPtr << ", " << *c_intPtr << '\n'; // outputs a
string, 10

    // *c_intPtr = 100; // error - cannot assign to a const pointer
    // *charPtr = 'A'; // error - cannot assign to a const pointer
}
```

`cend`

Get the sentinel at the end of the `const`-qualified range. The iterator can access the elements in the range, but can't modify them.

C++

```
template<class T>
constexpr std::sentinel_for<decltype(ranges::cbegin(std::declval<T>()))>
auto cend(T&& rg);
```

Parameters

T

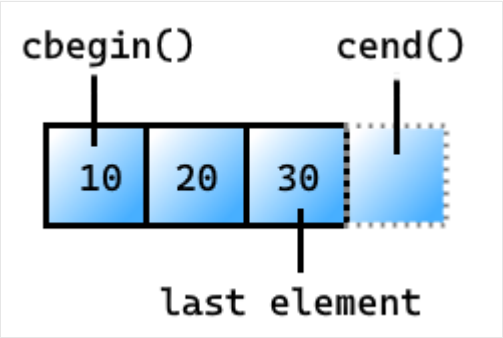
The type of the range.

rg

A range.

Return value

The sentinel that follows the last element in the `const`-qualified range:



Remarks

`ranges::cend()` works on all ranges, whereas `std::cend()` may not.

Example: `cend`

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> v = {10, 20, 30};
    auto i = std::ranges::cend(v);
    --i; // get off the sentinel and onto the last element in the range
    std::cout << *i; // outputs 30
    // *i = 300 // error because the iterator is const
}
```

`crbegin`

Get a reverse `const` iterator to the first element in a reversed range. A reverse iterator returns the elements of the range in reverse order. The range itself isn't reversed; access to it is.

C++

```
template<class T>
constexpr std::input_or_output_iterator auto crbegin(T&& rg);
```

Parameters

T

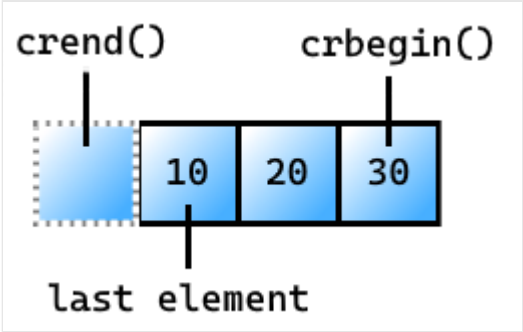
The type of the range.

rg

A range.

Return value

A reverse `const` iterator to the first element in the range. This iterator returns the elements of the range in reverse order, starting at the end of the range:



If the range is an array, returns the equivalent of `reverse_iterator{rg + n}` where `n` is the number of elements in the array. If `auto(rg.crbegin())` yields an iterator, returns the equivalent of `auto(rg.crbegin())`. If that expression is ill-formed, `auto(crbegin(rg))` is used if that expression yields an iterator.

Remarks

`ranges::crbegin()` works on all bidirectional ranges, whereas `std::crbegin()` may not.

Example: `crbegin`

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
```

```cpp
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    auto vi = std::ranges::crbegin(v);
    std::cout << *vi << ' ' << *++vi << ' ' << *++vi; // outputs 30 20 10
    // vi[1] = 100; // error because the iterator is const
}
```

`crend`

Get the sentinel at the end of what `crbegin()` returns. A reverse iterator returns the elements of the range in reverse order. The range itself isn't reversed; access to it is.

C++

```cpp
template<class T>
std::sentinel_for<decltype(ranges::crbegin(declval<T>()))> auto crend(T&&
rg);
```

Parameters

`T`

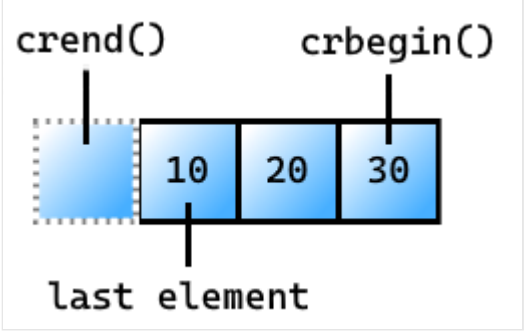
The type of the range.

`rg`

A range.

Return value

The sentinel at the end of what `cbegin()` returns. The sentinel follows the last element in a reversed view of the range:



Remarks

`ranges::crend()` works on all bidirectional ranges, whereas `std::crend()` may not.

`crend` example

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    auto vi = std::ranges::crend(v);
    --vi; // get off the sentinel and onto the last element in the reversed
range
    std::cout << *vi; // outputs 10
    // vi[0] = 300; // error because the iterator is const
    std::cout << *vi << ' ' << *--vi << ' ' << *--vi; // outputs 10, 20, 30
}
```

`data`

Get a pointer to the first element in a contiguous range.

C++

```
template<class T>
constexpr std::add_pointer_t<ranges::range_reference_t<T>> data(T&& rg);
```

Parameters

`T`

The type of the range.

`rg`

A range.

Return value

A pointer, based on the type of the range, to the first element in the contiguous range. For example, if the range is a vector of integers, the type of the return value is an `int *`.

Example

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    std::string src{ "a string" };

    auto charPtr = std::ranges::data(src); // charPtr is a char *
    auto intPtr = std::ranges::data(v); // intPtr is an int *
    std::cout << charPtr << ", " << *intPtr << '\n'; // outputs a string, 10
    *intPtr = 100;
    *charPtr = 'A';
    std::cout << charPtr << ", " << *intPtr; // outputs A string, 100
}
```

`empty`

Test if the range is empty.

C++

```
template<class T>
constexpr bool empty(T&& rg);
```

Parameters

`T`

The type of the range.

`rg`

A range.

Return value

Returns `true` if the range has no elements; otherwise `false.`

Example

```
C++
```

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
        std::vector v{10,20,30};
    std::vector<int> v2;

    std::cout << std::boolalpha << std::ranges::empty(v); // outputs false
    std::cout << std::boolalpha << ", " << std::ranges::empty(v2); //
outputs true
}
```

end

Get the sentinel at the end of the range.

```
C++
```

```
template<class T>
std::sentinel_for<ranges::iterator_t<T>> auto end(T&& rg);
```

Parameters

T

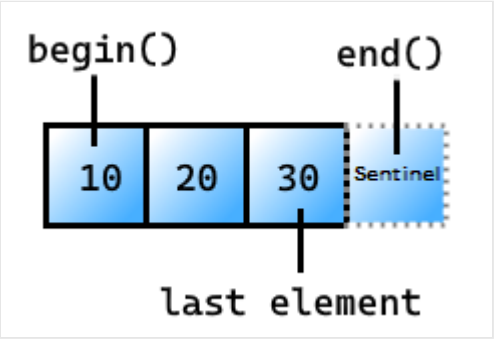
The type of the range.

rg

A range.

Return value

The sentinel that follows the last element in the range:



Remarks

`ranges::end()` works on all ranges, whereas `std::end()` may not.

Example

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> v = {10, 20, 30};
    auto i = std::ranges::end(v);
    --i; // get off the sentinel and onto the last element in the range
    std::cout << *i; // outputs 30
}
```

`rbegin`

Get a reverse iterator to the first element in a reversed range. A reverse iterator returns the elements of the range in reverse order. The range itself isn't reversed; access to it is.

C++

```
template<class T>
constexpr std::input_or_output_iterator auto rbegin(T&& rg);
```

Parameters

T

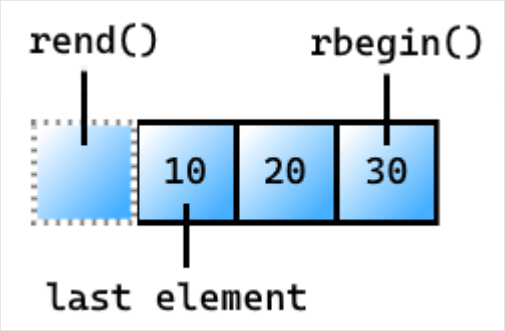
The type of the range.

`rg`

A range.

Return value

A reverse iterator to the first element in the range. This iterator returns the elements of the range in reverse order, starting at the end of the reversed range:



If the range is an array, returns the equivalent of `reverse_iterator{rg + n}` where `n` is the number of elements in the array. If `auto(rg.rbegin())` yields an iterator, returns the equivalent of `auto(rg.rbegin())`. If that expression is ill-formed, `auto(rbegin(rg))` is used if that expression yields an iterator.

Remarks

`ranges::rbegin()` works on all bidirectional ranges, whereas `std::rbegin()` may not.

Example

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    auto vi = std::ranges::rbegin(v);
    std::cout << *vi << ' ' << *++vi << ' ' << *++vi; // outputs 30 20 10
}
```

rend

Get a reverse iterator to the sentinel at the end of a reversed view of the range. A reverse iterator returns the elements of the range in reverse order. The range itself isn't reversed; access to it is.

```
C++
```

```
template<class T>
constexpr
std::sentinel_for<decltype(ranges::rbegin(std::declval<T>()))> auto rend(T&&
rg);
```

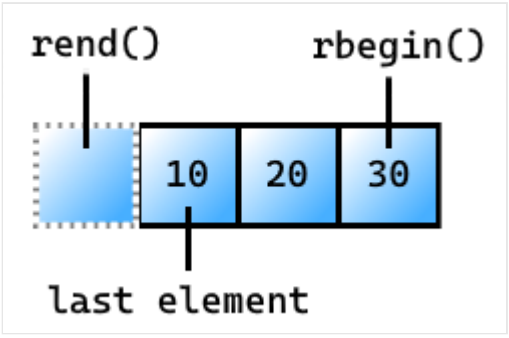
Parameters

T
The type of the range.

rg
A range.

Return value

A reverse iterator to the sentinel at the end of the range. The sentinel follows the last element in a reversed view of the range:



Remarks

`ranges::rend()` works on all bidirectional ranges, whereas `std::rend()` may not.

Example

```
C++
```

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
```

```
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    auto vi = std::ranges::rend(v);
    --vi; // get off the sentinel and onto the last element in the reversed
range
    std::cout << *vi; // outputs 10
    std::cout << *vi << ' ' << *--vi << ' ' << *--vi; // outputs 10, 20, 30
}
```

`size`

Get the number of elements in the range as an unsigned value.

C++

```
template<class T>
constexpr /*unsigned integer-like type */ size(T&& rg);
```

Parameters

T

The type of the range.

rg

A range.

Return value

The number of elements in the range as an unsigned integer-like value.

Remarks

This function executes in constant time.

Example

C++

```
// requires /std:c++20 or later
#include <vector>
```

```
#include <ranges>
#include <iostream>

int main()
{
    std::vector v{10, 20, 30};
    auto s = std::ranges::size(v); // s is a size_t
    std::cout << s; // outputs 3
}
```

`ssize`

Get the size of the range as a signed value.

C++

```
template<class T>
constexpr /* signed-integer-like type */ ssize(T&& rg);
```

Parameters

T

The type of the range.

rg

A range.

Return value

The number of elements in the range as a signed integer-like value.

Remarks

This function executes in constant time.

Example

C++

```
// requires /std:c++20 or later
#include <vector>
#include <ranges>
#include <iostream>
```

```
int main()
{
    std::vector v{10, 20, 30};
    auto s = std::ranges::ssize(v);
    std::cout << s; // outputs 3
}
```

See also

<ranges>

What are customization objects

`<ranges>` view classes

Article • 02/07/2023

A *view* is a lightweight range that refers to elements that it doesn't own (except owning_view). A view is typically based on another range and provides a different way of looking at it, whether by transforming or filtering it. For example, std::views::filter is a view that uses the criteria that you specify to select elements from another range.

When you access the elements in a view, it's done "lazily" so that work is done only when you get an element. This makes it possible to combine, or *compose*, views without a performance penalty.

For example, you could create a view that provides only the even elements from a range and then transform them by squaring them. The work to do the filtering and transformation is done only for the elements that you access, and only when you access them.

A view can be copied, assigned, and destroyed in constant time no matter how many elements it contains. This is because a view doesn't own the elements that it refers to, so it doesn't need to make a copy. This is why you can compose views without a performance penalty.

You typically create a view by using a range adaptor. Range adaptors are the intended way to create a view, are easier to use than instantiating the view classes directly, and are sometimes more efficient than instantiating the view classes directly. The view classes are exposed directly in case you need to create your own custom view type based on an existing view type.

Here's a brief example of creating a view of the squares of the elements that are divisible by three in a vector:

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> input =  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto divisible_by_three = [](const int n) {return n % 3 == 0;};
    auto square = [](const int n) {return n * n;};

    auto x = input
```

```
                | std::views::filter(divisible_by_three)
                | std::views::transform(square);

    for (int i : x)
    {
        std::cout << i << ' '; // 0 9 36 81
    }
}
```

Output

```
0 9 36 81
```

Using a view after the range that it's based on is modified can lead to undefined behavior. For example, a reverse_view based on a vector shouldn't be reused if you add or remove elements from the underlying vector. Modifying the underlying vector invalidates the container's `end` iterator--including the copy of the iterator that the view might have made.

Because views are cheap to create, you should generally re-create a view if you modify the underlying range. The following example demonstrates how to store a view pipeline in a variable so that you can reuse it.

C++

```
// requires /std:c++20, or later
#include <iostream>
#include <ranges>
#include <vector>
#include <list>
#include <string_view>
#include <algorithm>

template<typename rangeType>
void show(std::string_view msg, rangeType r)
{
    std::cout << msg;
    std::ranges::for_each(r,
        [](auto e)
        {
            std::cout << e << ' ';
        });
    std::cout << '\n';
}

int main()
{
    std::vector v{ 1, 2, 3, 4 };
    show("v: ", v);
```

```
    // You can save a view pipeline
    auto rev3 = std::views::take(3) | std::views::reverse;

    show("v | rev3: ", v | rev3); // 3 2 1

    v.insert(v.begin(), 0); // v = 0 1 2 3 4
    show("v: ", v);

    // Because modifying the vector invalidates its iterators, rebuild the
view.
    // We are reusing the view pipeline we saved earlier
    show("v | rev3(v): ", rev3(v));
}
```

Output

```
v: 1 2 3 4
v | rev3: 3 2 1
v: 0 1 2 3 4
v | rev3(v): 2 1 0
```

The following view classes are defined in the `std::ranges` namespace.

View	Description
basic_istream_view C++20	A view of successive elements from an input stream. Specializations include `istream_view` and `wistream_view`.
common_view C++20	Adapts a view that has different iterator/sentinel types into a view with the same iterator/sentinel types.
drop_view C++20	Created from another view, skipping the first `count` elements.
drop_while_view C++20	Created from another view, skipping leading elements as long as a predicate holds.
elements_view C++20	A view over the selected index into each tuple-like value in a collection. For example, given a range of `std::tuple<string, int>` values, create a view that consists of all the `string` elements from each tuple.
empty_view C++20	A view with no elements.
filter_view C++20	Filters out elements of a range that don't match a predicate.
iota_view C++20	A generated view that contains a sequence of incrementing values.
join_view C++20	Combines all the elements of multiple ranges into a single view.

View	Description
keys_view C++20	A view over the first index into each tuple-like value in a collection. For example, given a range of `std::tuple<string, int>` values, create a view that consists of the `string` elements from each tuple.
lazy_split_view C++20	Splits a view into subranges based on a delimiter.
owning_view C++20	Takes ownership of the elements from another range.
ref_view C++20	A view that references the elements that belong to another range.
reverse_view C++20	Presents the elements of a range in reverse order.
single_view C++20	A view that contains only one element.
split_view C++20	Splits a view into subranges based on a delimiter.
subrange C++20	A view of part of the elements of a range, as defined by a begin iterator and a sentinel.
take_view C++20	Contains the specified number of elements taken from the front of a range.
take_while_view C++20	Contains the leading elements of a range that match the given predicate.
transform_view C++20	A view of an underlying sequence after a transformation function is applied to each element.
values_view C++20	A view over the second index into each tuple-like value in a collection. For example, given a range of `std::tuple<string, int>` values, create a view that consists of the `int` elements from each tuple.

Many of these classes have corresponding range adaptors in the `std:views` namespace that creates instances of them. Prefer using an adaptor to create a view instead of creating view classes directly. The range adaptors are the intended way to create views, are easier to use, and in some cases are more efficient.

View classes characteristics

Each view class topic has a **Characteristics** section after the syntax section. The **Characteristics** section has the following entries:

- **Range adaptor**: A link to the range adaptor that creates the view. You typically use a range adaptor to create a view rather than create a view class directly, so it's listed here for convenience.

- **Underlying range**: Views have different iterator requirements for the kind of underlying range that they can use. See ranges iterator hierarchy for more information about the kinds of iterators.

- **View iterator category**: The iterator category of the view. When a view adapts a range, the iterator type for the view is typically the same as the iterator type of the underlying range. However, it might be different for some views. For example, `reverse_view` has a bidirectional_iterator, even if the underlying range has a random_access_iterator.

- **Element type**: The type of the elements that the view's iterator returns.

- **Sized**: Whether the view can return the number of elements that it refers to. Not all views can.

- **Common range**: Specifies whether the view is a common_range, which means that the begin iterator and sentinel types are the same. Common ranges are useful for pre-range code that works with iterator pairs. An example is iterator pair constructors for a sequence container, like `vector(ranges::begin(x), ranges::end(x))`.

- **Borrowed range**: Specifies whether the view is a borrowed range. `borrowed_range<T>` means you can use iterators for `T` after `T` is destroyed.

  No standard container is a borrowed range, because destroying the container frees the elements and invalidates any iterators. In that case, we say that the iterators are left "dangling" after destruction.

  For example, `std::ranges::find()` typically returns an iterator to the found element in the range argument. If the range argument is a temporary (rvalue) container, it's a mistake to store the returned iterator and use it later because it's "dangling."

  Range algorithms that return iterators (or subranges) do so only when their arguments are lvalues (non-temporaries) or borrowed ranges. Otherwise, they return a `std::dangling` object, which provides a hint in error messages about what went wrong if you tried to use it like an iterator.

- **Is `const` iterable**: Indicates whether you can iterate over a `const` instance of the view. Not all `const` views can be iterated. If a view isn't `const` iterable, you can't iterate with `for (const auto& element : as_const(theView))` or pass it to a function that takes a `const` reference to the view and then tries to iterate over it.

Ranges iterator hierarchy

In the **Characteristics** section of each view class topic, the iterator category in **Underlying range** and **View iterator category** refers to the kind of iterator that the range/view supports. There are six categories of Ranges iterators, which are identified by C++20 concepts. The hierarchy of range iterators, in increasing order of capability, is:

Range iterator concept	Description
output_range	Write-only, only moves forward; single-pass.
input_range	Read-only, only moves forward; single-pass.
forward_range	Only moves forward; multi-pass.
bidirectional_range	Can move forward and backward; multi-pass.
random_access_range	Can access the collection with an index; multi-pass.
contiguous_range	Can access the collection with an index, and elements are stored contiguously in memory.

Generally speaking, an iterator has the capability of the iterators that precede it in the table. For example, bidirectional_range has the capabilities of forward_range, but not vice versa. Except `input_range`, which doesn't have the capability of `output_range` because you can't write to an `input_range`.

The statement "requires `input_range` or higher" means that the view can be used with an `input_range`, `forward_range`, `bidirectional_range`, `random_access_range`, or `contiguous_range` iterator, because they're all as capable as `input_range`.

The ranges iterator hierarchy is directly related to the iterator hierarchy. For more information, see Iterator concepts.

See also

<ranges>
Range adaptors

`basic_istream_view` class (C++ Standard Library)

Article • 02/07/2023

A view of successive elements from an input stream.

Syntax

C++

```
template<movable Val, class CharT, class Traits>
    requires default_initializable<Val> &&
    stream_extractable <Val, CharT, Traits>
class basic_istream_view : public view_interface<basic_istream_view<Val,
CharT, Traits>>;
```

Template parameters

`CharT`

The character type of the stream.

`Traits`

Optional. Provides details about the character type of the stream regarding comparing characters, determining the length of a string made up of that character type, and so on. An example trait is char_traits<wchar_t>. If not specified, defaults to `char_traits<CharT>`.

`Val`

The type of the elements to extract. For example, `double` given a stream of: `"1.1 2.2 3.3"`

`stream_extractable` refers to the requirement (concept) that the type `Val` can be extracted from a stream using the `operator>>` function. It's equivalent to:

C++

```
template<class Val, class CharT, class Traits>
concept stream_extractable =
  requires(std::basic_istream<CharT,Traits>& is, Val& t) {
    is >> t;
  };
```

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	istream
Underlying range	None
Element type	The same as `Val`
View iterator category	input_range
Sized	No
Is `const`-iterable	No
Common range	No
Borrowed range	No

Specializations: `istream_view` and `wistream_view`

Convenience alias templates are provided for `char` and `wchar_t` streams, as follows:

C++

```
1) template<class Val>
using istream_view = ranges::basic_istream_view<Val, char>;

2) template<class Val>
using wistream_view = ranges::basic_istream_view<Val, wchar_t>;
```

1) Reads elements from an input stream composed of `char` characters.
2) Reads elements from an input stream composed of `wchar_t` characters.

For 1) and 2), `Val` refers to the type of the elements to extract. For example, `Val` is `double` given a stream of: `"1.1 2.2 3.3"`

Members

Member functions	Description
Constructors[C++20]	Construct a `basic_istream_view`.

Member functions	Description
begin C++20	Read the first value and get an iterator for the view.
end C++20	Returns `std::default_sentinel`

No member functions are inherited from `view_interface`.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `basic_istream_view`.

C++

```
constexpr explicit
basic_istream_view(std::basic_istream<CharT, Traits>& stream);
```

Parameters

`stream`
The stream to read from.

For information about the template parameter type, see Template parameters.

Return value

A `basic_istream_view` instance. The `basic_istream_view` internal stream pointer is initialized to `addressof(stream)`.

Remarks

The best way to create a `basic_istream_view` is by using the views::istream range adaptor. Range adaptors are the intended way to create view classes. The view types are

exposed in case you want to create your own custom view type.

Example: `basic_istream_view`, `istream_view`, and `wistream_view`

C++

```
// requires /std:c++20, or later
#include <ranges>
#include <iostream>
#include <sstream>

int main()
{
    // range adaptor
    std::istringstream streamOfdoubles{ "1.1 2.2 3.3 4.4 5.5" };
    for (const auto& elem : std::views::istream<double>(streamOfdoubles))
    {
        std::cout << elem << ' '; // 1.1 2.2 3.3 4.4 5.5
    }
    std::cout << '\n';

    // range adaptor - create a wistream_view
    std::wistringstream streamOfInts{ L"1 2 3 4 5" };
    for (const auto& elem : std::views::istream<int>(streamOfInts))
    {
        std::cout << elem << ' '; // 1 2 3 4 5
    }
    std::cout << '\n';

    // istream_view alias
    std::istringstream cpu1{ "8 0 8 0" };
    // equivalent std::ranges::istream_view<int, char>
    for (const auto& elem : std::ranges::istream_view<int>{cpu1})
    {
        std::cout << elem; // 8080
    }
    std::cout << '\n';

    // wistream_view alias
    std::wistringstream cpu2{ L"6 5 0 2" };
    // equivalent std::ranges::istream_view<int, wchar_t>
    for (const auto& elem : std::ranges::wistream_view<int>{cpu2})
    {
        std::cout << elem; // 6502
    }
    std::cout << '\n';

    // specify all template arguments
    std::wistringstream misc(L"S T L");
    std::ranges::basic_istream_view<wchar_t, wchar_t,
std::char_traits<wchar_t>> basic{misc};
```

```
    for (const auto& elem : basic)
    {
        std::wcout << elem << ' '; // S T L
    }
}
```

Output

```
1.1 2.2 3.3 4.4 5.5
1 2 3 4 5
8080
6502
S T L
```

begin

Read the first value and get an iterator to the view.

C++

```
constexpr auto begin();
```

Parameters

None.

Return value

An iterator pointing at the first element in the `basic_istream_view`.

end

Get the sentinel at the end of the view.

C++

```
constexpr std::default_sentinel_t end() const noexcept;
```

Parameters

None.

Return value

`default_sentinel_t`

See also

<ranges>
View classes

`common_view` class (C++ Standard Library)

Article • 02/07/2023

Take a range that may have different iterator and sentinel types and create a view that has the same iterator and sentinel type. This is useful for calling STL algorithms that accept ranges specified by iterator pairs.

Syntax

```
C++
```

```
template<ranges::view V>
    requires (!ranges::common_range<V> &&
std::copyable<ranges::iterator_t<V>>)
class common_view : public ranges::view_interface<common_view<V>>;
```

Template parameters

V

The type of the underlying view.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::common
Underlying range	Must satisfy forward_range or higher
Element type	Same as the underlying range
View iterator category	`forward_range` or random_access_range when the underlying range satisfies `random_access_range` and sized_range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range is `const` iterable

Characteristic	Description
Common range	Yes
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors C++20	Construct a `common_view`.
base C++20	Get the underlying view.
begin C++20	Get an iterator to the first element in the view.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements in the view.
Inherited from view_interface	**Description**
back C++20	Get the last element.
data C++20	Get a pointer to the first element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the view isn't empty.

Remarks

The best way to create a `common_view` is by using the views::common range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

This view is useful for passing a range that has different iterator/sentinel types to a legacy algorithm that expects them to be the same.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of a `common_view`.

```
C++

1) common_view() = default;
2) constexpr explicit common_view(V v);
```

Parameters

v
The underlying view.

For information about the template parameter type, see Template parameters.

Remarks

1) Default constructs the `common_view`.
2) Constructs a `common_view` from the underlying view using `std::move(v)`. An error will result if `V` is a common range to avoid misuse that would negatively impact performance.

base

Gets a copy of the underlying view.

```
C++

// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
constexpr V base() &&;
```

`begin`

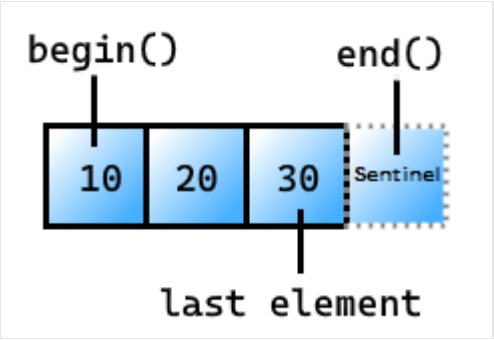
Get an iterator to the first element.

```
C++
```

```
constexpr auto begin();
constexpr auto begin() const requires range<const V>;
```

Return value

An iterator pointing at the first element in the view:



`end`

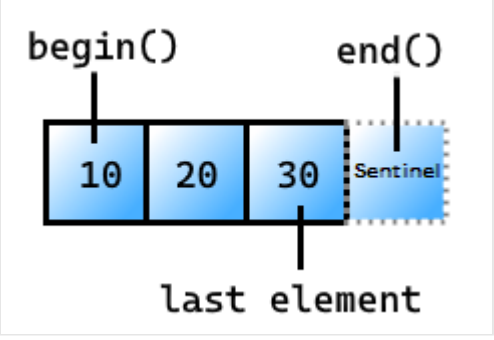
Get the sentinel at the end of the view.

```
C++
```

```
constexpr auto end();
constexpr auto end() const requires ranges::range<const V>;
```

Return value

The sentinel that follows the last element in the view:



`size`

Get the number of elements in the view.

```
C++
```

```
constexpr auto size() requires ranges::sized_range<V>;
constexpr auto size() const requires ranges::sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the view.

See also

<ranges>
common range adaptor
view classes

`drop_view` class (C++ Standard Library)

Article • 02/07/2023

Create a view that excludes the first *N* elements of a range.

Syntax

```
C++
```

```
template<ranges::view V>
class drop_view : public ranges::view_interface<drop_view<V>>;
```

Template parameters

V

The type of the underlying view.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::drop
Underlying range	Must satisfy output_range or higher
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range is `const` iterable and satisfies random_access_range and sized_range
Common range	Only if the underlying range is a common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors C++20	Construct a `drop_view`.
base C++20	Get the underlying view.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements in this view. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back C++20	Get the last element.
data C++20	Get a pointer to the first element.
empty C++20	Test whether the `drop_view` is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the `drop_view` isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `drop_view`

C++

```
template<ranges::view V>
class drop_view : public ranges::view_interface<drop_view<V>>
```

Template parameters

V

The type of the underlying view.

Return value

A view of the underlying range, excluding the specified number of elements from the front.
If you specify more elements to drop than exist in the underlying range, then an `empty_view` is returned.

Remarks

The best way to create a `drop_view` is by using the views::drop range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

Example: `drop_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{ 1, 2, 3, 4, 5 };
    auto newView = std::views::drop(v, 3);
    for (auto e : newView) // outputs 4 5
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';

    auto numbers = std::views::iota(0) | std::views::take(10); // generate a
view of 10 integers
    for (auto i : numbers | std::views::drop(5)) // use the '|' syntax to
create a drop_view
    {
        std::cout << i << ' '; // outputs 5 6 7 8 9
    }
}
```

Output

```
4 5
5 6 7 8 9
```

`base`

Gets a copy of the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
constexpr V base() &&;
```

Parameters

None.

Return value

The underlying view.

`begin`

Get an iterator to the first element in the `drop_view`.

C++

```
constexpr auto begin()
  requires (!(Simple_view<V> && ranges::random_access_range<const V> &&
ranges::sized_range<const V>));

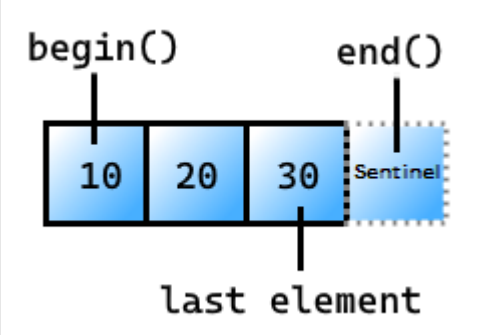
constexpr auto begin() const
  requires ranges::random_access_range<const V> && ranges::sized_range<const
V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the `drop_view`.



`end`

Get the sentinel at the end of the `drop_view`

```
C++
```

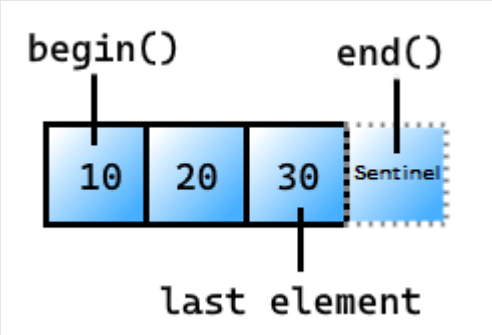
```
constexpr auto end() requires (!Simple_view<V>);
constexpr auto end() const requires ranges::range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `drop_view`:



`size`

Get the number of elements in the `drop_view`.

```
```

```
C++
```

```
constexpr auto size() requires ranges::sized_range<V>;
constexpr auto size() const requires ranges::sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the `drop_view`.

Remarks

The underlying range must satisfy sized_range.

See also

<ranges>
drop range adaptor
take_while()
take_while_view
View classes

`drop_while_view` class (C++ Standard Library)

Article • 02/07/2023

A view that contains the elements of a range that remain once the leading elements that match a predicate are dropped.

Syntax

```
C++
```

```
template<ranges::view V, class P>
    requires ranges::input_range<V> &&
             std::is_object_v<P> &&
             std::indirect_unary_predicate<const P, ranges::iterator_t<V>>
class drop_while_view : public ranges::view_interface<drop_while_view<V,
P>>;
```

Template parameters

V

The type of the underlying view.

P

The type of the predicate that determines the leading elements to drop.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::drop_while
Underlying range	Must satisfy forward_range or higher and the underlying range's iterators must model sized_sentinel_for
Element type	Same as the underlying range
View iterator category	Same as the underlying range

Characteristic	Description
Sized	Only if the underlying range satisfies random_access_range
Is `const`-iterable	No
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors	Construct the view.
base	Get the underlying view.
begin	Get an iterator to the first element.
end	Get the sentinel at the end of the view.
pred	Get a reference to the predicate that determines which elements to drop.
Inherited from view_interface	**Description**
back C++20	Get the last element.
data C++20	Get a pointer to the first element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the view isn't empty.
size	Get the number of elements in the view.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `drop_while_view`.

```
C++
```

```
1) constexpr drop_while_view(V base, P pred);
2) drop_while_view() requires default_initializable<V> &&
default_initializable<P> = default;
```

Parameters

`base`
The underlying range.

`pred`
The predicate that determines the leading elements to drop.

For information about the template parameter types, see Template parameters.

Return value

A `drop_while_view` instance.

Remarks

The best way to create a `drop_while_view` is by using the views::drop_while range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Move constructs the `drop_while_view` from a *`base`* view and a *`pred`* predicate. Both *`base`* and *`pred`* are moved via `std::move()`.
2) Default-constructs a `drop_while_view`.

Example: `drop_while_view`

```
C++
```

```
// requires /std:c++20 or later
#include <ranges>
#include <vector>
#include <iostream>

void print(auto v)
{
    for (auto& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto myView = std::views::drop_while(
        v,
        [](int i) {return i >= 0; });
    print(myView); // -4 5 6

    auto myView2 = v | std::views::drop_while(
        [](int i) {return i < 5; });
    print(myView2); // 5 6
}
```

Output

```
-4 5 6
5 6
```

base

Get the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses std::move() to return the underlying view
constexpr V base() &&;
```

Parameters

None.

Returns

The underlying view.

`begin`

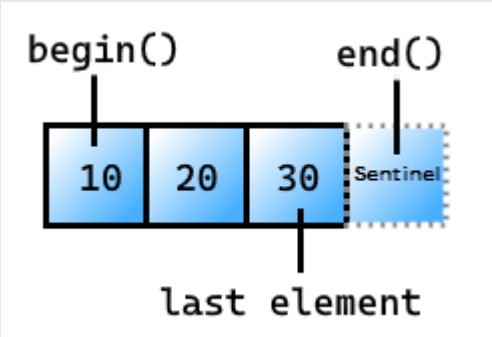
Get an iterator to the first element in the view.

```
C++
```

```
constexpr auto begin();
```

Return value

An iterator pointing at the first element in the view. The behavior is undefined if the view doesn't have a predicate.



`end`

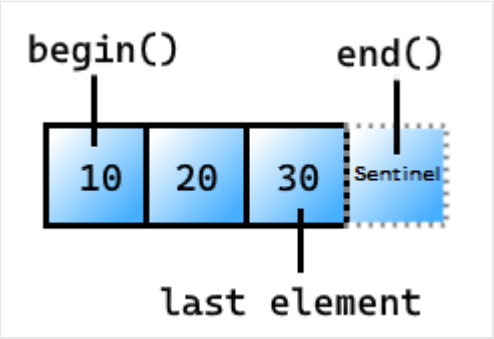
Get the sentinel at the end of the view.

```
C++
```

```
constexpr auto end()
```

Return value

The sentinel that follows the last element in the view:



`pred`

Get a reference to the predicate that determines which leading elements to drop.

C++

```
constexpr const Pred& pred() const;
```

Return value

A reference to the predicate.

Remarks

If the view doesn't store a predicate, the behavior is undefined.

Example `pred`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto mv = v | std::views::drop_while(
        [](int i) {return i < 5; }); // drop the leading elements < 5
    std::cout << std::boolalpha << mv.pred()(v[6]); // outputs "false"
because v[6] = 6 and 6 is not less than 5 (the predicate)
}
```

Output

```
false
```

See also

<ranges>
drop_while range adaptor
take_while range adaptor
take_while_view
view classes

`elements_view` class (C++ Standard Library)

Article • 02/07/2023

A view over the elements at a selected index in each tuple-like value in a range. For example, given a range of `std::tuple<string, int>`, create a view consisting of the `string` elements from each tuple.

Syntax

C++

```
template<input_range V, size_t N>
class elements_view : public view_interface<elements_view<V, N>>;
```

Template parameters

`N`

The index of the element to select for the view.

`V`

The type of the underlying range. This type must satisfy `ranges::input_range`.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::elements
Underlying range	Must satisfy input_range or higher
Element type	Same as the type of the indexed tuple element
View iterator category	forward_range, bidirectional_range, or random_access_range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range satisfies `const-iterable`

Characteristic	Description
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors C++20	Construct a `elements_view`.
base C++20	Get the underlying range.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements in this view. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back C++20	Get the last element.
empty C++20	Test whether the `elements_view` is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the `elements_view` isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Remarks

The tuple-like types that you can use with `elements_view` are std::tuple, std::pair, and std::array.

Constructors

Construct an instance of a `elements_view`.

```
C++

1) constexpr elements_view(V base);
2) elements_view() requires std::default_initializable<V> = default;
```

Parameters

`base`

The underlying range.

For information about the template parameter type, see Template parameters.

Return value

An `elements_view` instance.

Remarks

The best way to create an `elements_view` is by using the elements range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Create an `elements_view` from the specified view.
2) Default construct an `elements_view`.

Example: `elements_view`

```
C++

// requires /std:c++20 or later
#include <array>
#include <iostream>
#include <map>
#include <ranges>
#include <vector>
#include <string>
#include <utility>

int main()
{
```

```
    // ========== work with a std::map

    std::map<std::string, int> cpp_standards
    {
        {"C++98", 1988},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"c++20", 2020}
    };

    // create an elements_view of all the string elements (<1>) from each
tuple
    for (int const year : std::views::elements<1>(cpp_standards))
    {
        std::cout << year << ' '; // 2003 2011 2014 2017 1988 2020
    }

    std::cout << '\n';

    // Another way to call the range adaptor using pipe (|) syntax
    for (auto&& name : cpp_standards | std::views::elements<0>)
    {
        std::cout << name << ' '; // C++03 C++11 C++14 C++17 C++98 c++20
    }
    std::cout << '\n';

    // ========== working with arrays

    std::array<std::array<int, 4>, 3> arr = { {{0,1,2,3}, {4,5,6,7},
{8,9,10,11}} };
    for (int& fourth : arr | std::views::elements<3>)
    {
        std::cout << fourth << ' '; // 3 7 11
    }
    std::cout << '\n';

    // ========== work with a std::pair

    std::vector<std::pair<std::string, int>> windows
    {
        {"Windows 1.0", 1985},
        {"Windows 2.0", 1987},
        {"Windows 3.0", 1990},
        {"Windows 3.1", 1992},
        {"Windows NT 3.1", 1993},
        {"Windows 95", 1995},
        {"Windows NT 4.0", 1996},
        {"Windows 98", 1998},
        {"Windows 2000", 2000}
    };

    for (int year : std::views::elements<1>(windows))
    {
```

```
        std::cout << year << ' '; // 1985 1987 1990 1992 1993 1995 1996 1998
2000
    }
}
```

Output

```
2003 2011 2014 2017 1988 2020
C++03 C++11 C++14 C++17 C++98 c++20
3 7 11
1985 1987 1990 1992 1993 1995 1996 1998 2000
```

`base`

Gets a copy of the underlying range.

C++

```
// Uses a copy constructor to return the underlying range
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying range
constexpr V base() &&;
```

Parameters

None.

Return value

The underlying range.

`begin`

Get an iterator to the first element in the `elements_view`.

C++

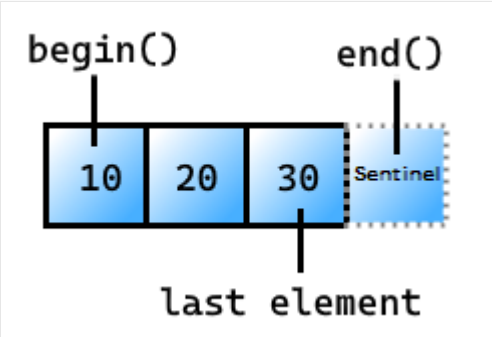
```
1) constexpr auto begin() requires (!Simple_view<V>);
2) constexpr auto begin() const requires range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the `elements_view`.



end

Get the sentinel at the end of the `elements_view`

```
C++
```

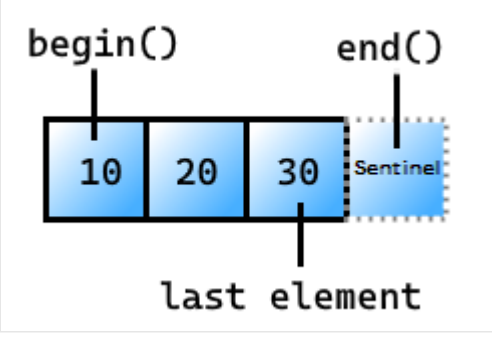
```
1) constexpr auto end() requires (!Simple_view<V> &&
!ranges::common_range<V>);
2) constexpr auto end() requires (!Simple_view<V> &&
ranges::common_range<V>);
3) constexpr auto end() const requires ranges::range<const V>;
4) constexpr auto end() const requires ranges::common_range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `elements_view`:



`size`

Get the number of elements in the view.

```
C++
```

```
constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the `elements_view`.

Remarks

The size of the view is only available if the underlying range is a sized_range, or in other words, bounded.

See also

keys_view
values_view
View classes
<ranges>

`empty_view` class (C++ Standard Library)

Article • 02/07/2023

A view with no elements. This view is useful for test purposes such as calling code that needs to be provided with a view but doesn't need to access its underlying data.

Syntax

C++

```
template<class T>
    requires std::is_object_v<T>
class empty_view : public ranges::view_interface<empty_view<T>>;
```

Template parameters

T

The type of the element. Even though there are no elements in an `empty_view`, all ranges are homogeneous. That is, they have elements of a particular type. So even though an `empty_view` has no elements, it still has a type, such as an `empty_view` of `int`, or `strings`, etc.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::empty
Underlying range	None
Element type	As specified when the `empty_view` is created
View iterator category	`contiguous_range`
Sized	Yes. Always returns 0
Is `const`-iterable	Yes
Common range	Yes

Characteristic	Description
Borrowed range	Yes

Members

Member functions	Description
Constructors C++20	Construct an `empty_view`.
begin C++20	Returns `nullptr`.
end C++20	Returns `nullptr`.
size C++20	Returns 0
Inherited from view_interface	**Description**
back C++20	Results in undefined behavior.
data C++20	Returns `nullptr`.
empty C++20	Returns `true`.
front C++20	Results in undefined behavior.
operator[] C++20	Results in undefined behavior.
operator bool C++20	Returns `false`.

Remarks

The best way to create a `empty_view` is by using the empty range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

Because there can never be any elements in an `empty_view`, certain compiler optimizations are possible. For example, the compiler will eliminate `for (auto e : std::views::empty<int>) {...}` because it knows that there's nothing to iterate over.

Another use for `empty_view` is splitting a split_view with an `empty_view` delimiter, which results in a range of single element ranges.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of a `empty_view`.

C++

```
template<class T>
inline constexpr empty_view<T> empty{};
```

Parameters

T

The type of the underlying element, of which there is none.

Remarks

The best way to create a `empty_view` is by using the empty range adaptor.

Example `empty_view`

C++

```
// requires /std:c++20 or higher
#include <ranges>
#include <iostream>

int main()
{
    auto anEmptyView = std::views::empty<int>;
    bool isNotEmpty = (bool)anEmptyView;
    std::cout << std::boolalpha << isNotEmpty << "\n"; // false
    std::cout << std::boolalpha << anEmptyView.empty(); // true
}
```

Output

```
false
true
```

`back`

Results in undefined behavior.

```
C++
```

```
constexpr auto back()
    requires ranges::bidirectional_range<T> && ranges::common_range<T>;

constexpr auto back() const
    requires ranges::bidirectional_range<const T> &&
ranges::common_range<const T>;
```

Parameters

None.

Return value

None.

Remarks

Calling this function in a debug build raises an assertion that the function has been called on an empty `view_interface`.

`begin`

Returns `nullptr` because there isn't a first element in the view.

```
C++
```

```
static constexpr T* begin() noexcept
```

Return value

`nullptr`

`data`

Returns `nullptr` because there isn't a first element in the view to get a pointer to.

```
C++
```

```
static constexpr T* data() noexcept
```

Return value

`nullptr`.

`empty`

Test whether the derived view is empty.

```
C++
```

```
static constexpr bool empty() noexcept
```

Parameters

None.

Return value

Returns `true`.

`end`

Returns `nullptr` because there aren't any elements in the view.

```
C++
```

```
static constexpr T* end() noexcept
```

Return value

`nullptr`.

`front`

Results in undefined behavior.

```
C++
```

```
constexpr auto front()
    requires ranges::forward_range<T>;
constexpr auto front() const
    requires ranges::forward_range<const T>;
```

Parameters

None.

Return value

None.

Remarks

Calling this function in a debug build raises an assertion that the function has been called on an empty `view_interface`.

`operator[]`

Results in undefined behavior.

```
C++
```

```
template<ranges::random_access_range R = T>
constexpr decltype(auto) operator[](ranges::range_difference_t<R> pos);

template<ranges::random_access_range R = const T>
constexpr decltype(auto) operator[](ranges::range_difference_t<R> pos)
const;
```

Parameters

pos

The position, relative to the beginning iterator, of the element to return.

Return value

None.

Remarks

Calling this function in a debug build raises an assertion that index is out of range for `view_interface`.

`operator bool`

Test whether the derived view isn't empty.

```
C++
```

```
constexpr explicit operator bool()
requires requires { ranges::empty(T ()); };

constexpr explicit operator bool() const
requires requires { ranges::empty(T ()); };
```

Parameters

None.

Return value

Returns `false`.

Example `(bool)`

```
C++
```

```
// requires /std:c++20 or higher
#include <ranges>
#include <iostream>

int main()
{
    auto anEmptyView = std::views::empty<int>;

    if (anEmptyView) // check if anEmptyView isn't empty
    {
        std::cout << "Error: why does an empty_view have elements?\n";
```

```
    }
    else
    {
        std::cout << "Correct: an empty_view is not not empty\n";
    }
}
```

Output

```
Correct: an empty_view is not not empty
```

`size`

Get the number of elements in the view, which will always be 0.

C++

```
static constexpr size_t size()
```

Parameters

None.

Return value

`0`.

See also

<ranges>
empty range adaptor
single_view
view classes

`filter_view` class (C++ Standard Library)

Article • 02/07/2023

A view that filters out the elements of a range that don't match the predicate.

Syntax

C++

```
template<ranges::input_range V, indirect_unary_predicate<iterator_t<V>>
Pred>
    requires view<V> && is_object_v<Pred>
class filter_view : public view_interface<filter_view<V, Pred>>;
```

Template parameters

`V`

The type of the underlying range.

`Pred`

The type of the predicate that determines which elements to keep.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::filter
Underlying range	Must satisfy input_range or higher
Element type	Same as the underlying range
View iterator category	`input_range`, forward_range, or bidirectional_range depending on the underlying range
Sized	No
Is `const`-iterable	No

Characteristic	Description
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors C++20	Construct the view.
base C++20	Get the underlying range.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
pred C++20	Get a reference to the predicate that determines which elements to drop.
Inherited from view_interface	**Description**
back C++20	Get the last element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator bool C++20	Test whether the view isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `filter_view`

```
C++
```

```
1) constexpr filter_view(V base, P pred);
2) filter_view() requires default_initializable<V> &&
default_initializable<Pred> = default;
```

Parameters

base

The underlying view.

pred

The predicate that determines which elements to keep from the underlying view.

For information about the template parameter types, see Template parameters.

Return value

A `filter_view` instance.

Remarks

The best way to create a `filter_view` is by using the views::filter range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Create a value-initialized `filter_view`. The predicate and the underlying view must be default-initializable.
2) Move constructs the `filter_view` from a *base* view and a *pred* predicate. Both *base* and *pred* are moved via `std::move()`.

Example: `filter_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

void print(auto v)
{
    for (auto& x : v)
    {
        std::cout << x << ' ';
```

```
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{0, 1, -2, 3, -4, -5, 6};
    auto myView = std::views::filter(v, [](int i) {return i > 0; });
    print(myView); // outputs 1 3 6

    auto myView2 = v | std::views::filter([](int i) {return i < 3; });
    print(myView2); // outputs 0 1 -2 -4 -5
}
```

Output

```
1 3 6
0 1 -2 -4 -5
```

`base`

Gets the underlying range.

C++

```
// Uses a copy constructor to return the underlying range
constexpr V base() const& requires std::copy_constructible<V>;

// Uses std::move() to return the underlying range
constexpr V base() &&;
```

Parameters

None.

Returns

The underlying view.

`begin`

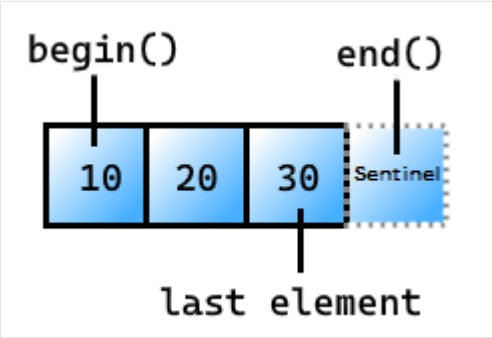
Get an iterator to the first element in the view.

C++

```
constexpr auto begin();
```

Return value

An iterator pointing at the first element in the view. The behavior is undefined if the view doesn't have a predicate.



`end`

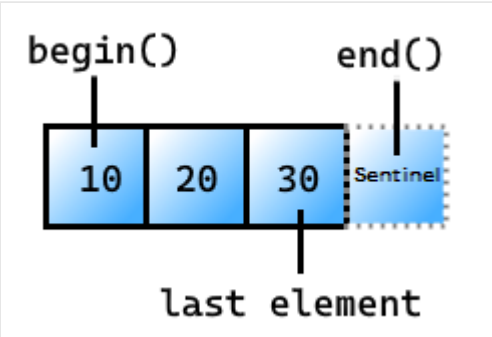
Get the sentinel at the end of the view.

C++

```
constexpr auto end()
```

Return value

The sentinel that follows the last element in the view:



`pred`

Get a reference to the predicate that determines which leading elements to drop.

C++

```
constexpr const Pred& pred() const;
```

Return value

A reference to the predicate.

Remarks

If the class doesn't store a predicate, the behavior is undefined.

Example `pred`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};
    auto mv = v | std::views::filter(
        [](int i) {return i < 5; }); // keep the elements < 5

    std::cout << std::boolalpha << mv.pred()(v[6]); // outputs "false"
because v[6] = 6 and 6 is not less than 5 (the predicate)
}
```

See also

<ranges>
filter range adaptor
drop_while()
take_while()
view classes

`iota_view` class (C++ Standard Library)

Article • 02/07/2023

Generates a view of a sequence of elements by repeatedly incrementing an initial value. The sequence can be bounded or unbounded.

Syntax

C++

```
template<std::weakly_incrementable W, std::semiregular Bound =
std::unreachable_sentinel_t>
    requires __WeaklyEqualityComparableWith<W, Bound> && std::copyable<W>
class iota_view : public ranges::view_interface<iota_view<W, Bound>>;
```

Template parameters

`W`

The type of the values in the sequence. The specified type must support `operator++`.

`Bound`

The type of the end value. If *`Bound`* is `std::unreachable_sentinel_t` (the default value), then the view is unbounded.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::iota
Underlying range	Not applicable because this view generates its members
Element type	Same as the template parameter `W`
View iterator category	Supports input_range up to random_access_range, depending on the type of `W`
Sized	Only if the range has an end value, that is, it isn't an infinite series
Is `const`-iterable	Yes

Characteristic	Description
Common range	Only if `Bound` is the same type as `W`
Borrowed range	Yes

Members

Member functions	Description
Constructors C++20	Construct the view.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements.
Inherited from view_interface	**Description**
back C++20	Get the last element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the view isn't empty.

Remarks

A good way to create a `iota_view` is by using the iota range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

The sequence of values can be integral values such as 1,2,3 or 'a', 'b', 'c' or consecutive elements from a range.

This view is typically used to iterate over a series of values. For example:

C++

```
for (int i : iota_view{1, 10}) // iterate over a view of the integers 1
through 9
{
```

```
    std::cout << i << ' '; // 1 2 3 4 5 6 7 8 9
}
```

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of an `iota_view`.

C++

```
1) iota_view() requires std::default_initializable<W> = default;
2) constexpr explicit iota_view(W value);
3) constexpr iota_view(std::type_identity_t<W> value,
std::type_identity_t<Bound> bound);
4) constexpr iota_view( /*iterator*/ first, /*sentinel*/ last );
```

Parameters

value

The starting value for the series.

bound

The bound of the series. It's one greater than the last value in the series.

For information about the template parameter type, see Template parameters.

Remarks

1) Create an `iota_view` with a starting and ending value determined by the default-initialized template type `W`.
2) Create an `iota_view` with the specified starting value and ending value determined by the value-initialized type `W`.
3) Create a bounded `iota_view` with the specified starting value and ending value. The ending value is one less than the last value specified.

4) Used to create subviews. For example, `std::ranges::iota_view(start, end);` where `start` and `end` are iterators to the start and end of the subview.

Rather than create this class directly, an easy way to create a `iota_view` is by using the iota range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

If the bound's type is `unreachable_sentinel_t`, the resulting view is bounded.

When you use one of the constructors that value-initializes the bound, the bound will be the default-initialized value for that type. For example, `iota_view<int, int>{}` is an empty range of `int` since the default-constructed int value and bound are both `0`. Also, `iota_view<int, int>{-4}` is `-4, -3, -2, -1` because the default-constructed `int` bound is `0`.

`iota_view` constructor example

C++

```
// requires /std:c++20, or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::ranges::iota_view<int> iv; // create an iota_view with an unbounded
range, starting at 0
    std::ranges::iota_view<int> iv2(5); // create an iota_view with an
unbounded range, starting at 5.
    std::ranges::iota_view<int, int> iv3{5, 10}; // create an iota_view with
a bounded range, starting at 5 and ending at 9

    std::vector<int> v{10, 20, 35, 45, 50, 66, 77, 82, 90, 100};
    auto start = std::ranges::find(v, 35);
    auto end = std::ranges::find(v, 82);
    for (auto &&val : std::ranges::iota_view(start, end))
    {
        std::cout << *val << ' '; // outputs 35 45 50 66 77
    }
}
```

`begin`

Get an iterator to the first element in the view.

C++

```
constexpr auto begin();
constexpr auto begin() const requires range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the view.

`end`

Get the end sentinel at the end of the view.

C++

```
constexpr auto end();
constexpr auto end() const requires ranges::range<const V>;
```

Parameters

None.

Return value

If the `iota_view` is unbounded, returns `std::unreachable_sentinel`.

If the `iota_view` is bounded, returns an iterator pointing at the sentinel past the last value in the sequence.

`size`

Get the number of elements in the view. The `iota_view` must be bounded.

C++

```
constexpr auto size() requires
    (same_as<W, Bound> && advanceable <W>) ||
```

```
    (integral<W> && integral<Bound>) ||
    sized_sentinel_for<Bound, W>;
```

Parameters

None.

Return value

The number of elements in the `iota_view`.

Remarks

You can't call `size()` on an unbounded `iota_view`.

See also

<ranges>
iota() range adaptor
std::numerics::iota
view classes

`join_view` class (C++ Standard Library)

Article • 02/07/2023

Combines the elements of a range of ranges into a single view.

Syntax

```
C++
```

```
template<input_range R> requires view<R> &&
input_range<range_reference_t<R>> &&
    (is_reference_v<range_reference_t<R>> || view<range_value_t<R>>)
class join_view : public view_interface<join_view<R>>;
```

Template parameters

R

The type of the underlying range. Must satisfy input_range or higher.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::join
Underlying range	Must satisfy input_range or higher
Element type	Same as the underlying range
View iterator category	`input_range` up to bidirectional_range depending on the underlying range being iterated
Sized	No
Is `const`-iterable	Only if the underlying range is `const` iterable
Common range	Only if the underlying range satisfies common_range
Borrowed range	No

Members

Member functions	Description
Constructors C++20	Construct an `join_view`.
base C++20	Get a reference to the underlying range.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
Inherited from view_interface	**Description**
back C++20	Get the last element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator bool C++20	Test whether the view isn't empty.

Remarks

The best way to create a `join_view` is by using the views::join range adaptor. Range adaptors are the intended way to access view classes. The view types are exposed in case you want to create your own custom view type.

This view is useful when you want to combine multiple ranges into a single view.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of a `join_view`.

```
C++
```

```
1) join_view() = default;
2) constexpr explicit join_view(R base)
```

Parameters

base

The underlying range.

For information about the template parameter type, see Template parameters.

Remarks

1) Default-constructs a `join_view`.
2) Constructs the `join_view` from *base*.

Example: `join_view`

C++

```
// requires /std:c++20, or later
#include <iostream>
#include <vector>
#include <ranges>

int main()
{
    std::vector<int> rg1{1, 2, 3, 4};
    std::vector<int> rg2{5, 6, 7};
    std::vector<int> rg3{8, 9, 10, 11, 12, 13};
    std::vector<int> rangeOfRanges[] {rg1, rg2, rg3};

    auto jv = std::ranges::join_view(rangeOfRanges);

    for (auto& e : jv)
    {
        std::cout << e << " ";
    }
}
```

Output

```
1 2 3 4 5 6 7 8 9 10 11 12 13
```

`base`

Get the underlying view.

```
C++
```

```
// Uses a copy constructor to return the underlying view
1) constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
2) constexpr V base() &&;
```

Parameters

None.

Return value

1. The returned view is copy constructed.
2. The returned view is move constructed.

`begin`

Get an iterator to the first element in the view.

```
C++
```

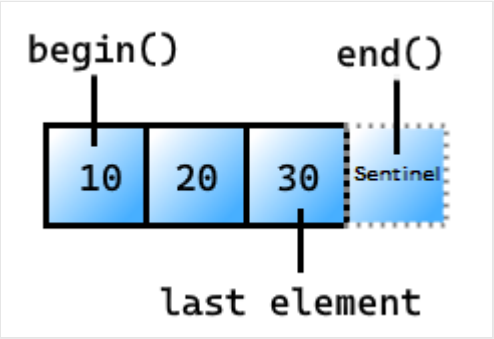
```
constexpr auto begin();
constexpr auto begin() const
    requires ranges::input_range<const V> &&
std::is_reference_v<ranges::range_reference_t<const V>>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the view:



end

Get the sentinel at the end of the view.

```
C++
```

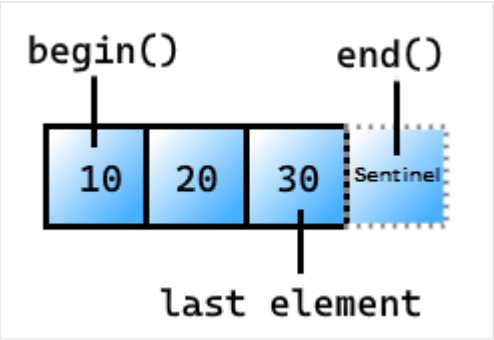
```
constexpr sentinel_t<R> end();
constexpr auto end() const requires range<const R>
```

Parameters

None.

Return value

The sentinel that follows the last element in the view:



See also

join range adaptor
<ranges>
view classes

`keys_view` class (C++ Standard Library)

Article • 02/07/2023

A view over the first index into each tuple-like value in a collection. For example, given a range of `std::tuple<string, int>`, create a view consisting of all the `string` elements from each tuple.

`keys_view` is an alias for elements_view<R, 0> and is useful for making of view of the keys from associative containers like std::map or std::unordered_map.

Syntax

C++

```
template<input_range R>
using keys_view = ranges::elements_view<R, 0>;
```

Template parameters

`R`

The type of the underlying range. This type must satisfy `ranges::input_range`.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::keys
Underlying range	Must satisfy input_range or higher
Element type	Same as the type of the first tuple element of the underlying range
View iterator category	random_access_range if the underlying range is contiguous, otherwise the same as the underlying range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range satisfies `const-iterable`

Characteristic	Description
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

The following list of member functions refers to the `keys_view` class. Recall that this is an alias for an element_view class template instantiation.

Member functions	Description
Constructors[C++20]	Construct a `keys_view`.
base[C++20]	Get the underlying range.
begin[C++20]	Get an iterator to the first element.
end[C++20]	Get the sentinel at the end of the view.
size[C++20]	Get the number of elements. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
empty[C++20]	Test whether the `keys_view` is empty.
front[C++20]	Get the first element.
operator[][C++20]	Get the element at the specified position.
operator bool[C++20]	Test whether the `keys_view` isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Remarks

The tuple-like types that you can use with `keys_view` are std::tuple, std::pair, and std::array.

Constructors

Construct an instance of a `keys_view`.

```
C++
```

```
1) constexpr keys_view(V base);
2) keys_view() requires std::default_initializable<V> = default;
```

Parameters

`base`
The underlying range.

For information about the template parameter type, see Template parameters.

Return value

A `keys_view` instance.

Remarks

The best way to create an `keys_view` is by using the keys range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Create a `keys_view` from the specified view.
2) The default constructor creates an empty `keys_view`.

Example: `keys_view`

```
C++
```

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <map>
#include <vector>
#include <string>
```

```cpp
#include <utility>

int main()
{
    // ========== work with a std::map

    std::map<std::string, int> cpp_standards
    {
        {"C++98", 1988},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"c++20", 2020}
    };

    // Extract all of the keys from the map
    for (const std::string& standards : std::views::keys(cpp_standards))
    {
        std::cout << standards << ' '; // C++03 C++11 C++14 C++17 C++98
c++20
    }
    std::cout << '\n';

    // ========== work with a range of std::pair

    std::vector<std::pair<std::string, int>> windows
    {
        {"Windows 1.0", 1985},
        {"Windows 2.0", 1987},
        {"Windows 3.0", 1990},
        {"Windows 3.1", 1992},
        {"Windows NT 3.1", 1993},
        {"Windows 95", 1995},
        {"Windows NT 4.0", 1996},
        {"Windows 95", 1995},
        {"Windows 98", 1998},
        {"Windows 1.0", 1985},
        {"Windows 2000", 2000}
    };

    // Another way to call the range adaptor using '|': create an keys_view
from each pair
    for (const std::string& version : windows | std::views::keys)
    {
        std::cout << version << ' '; // Windows 1.0 Windows 2.0 Windows 3.0
...
    }
}
```

Output

```
C++03 C++11 C++14 C++17 C++98 c++20
Windows 1.0 Windows 2.0 Windows 3.0 Windows 3.1 Windows NT 3.1 Windows 95
Windows NT 4.0 Windows 95 Windows 98 Windows 1.0 Windows 2000
```

`base`

Gets a copy of the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
constexpr V base() &&;
```

Parameters

None.

Return value

The underlying view.

`begin`

Get an iterator to the first element in the `keys_view`.

C++

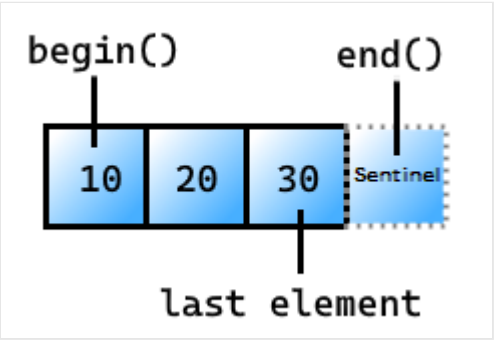
```
1) constexpr auto begin() requires (!Simple_view<V>);
2) constexpr auto begin() const requires (Simple_view<V>) // or put another
way, requires ranges::range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the `keys_view`.



end

Get the sentinel at the end of the `keys_view`

```
C++
```

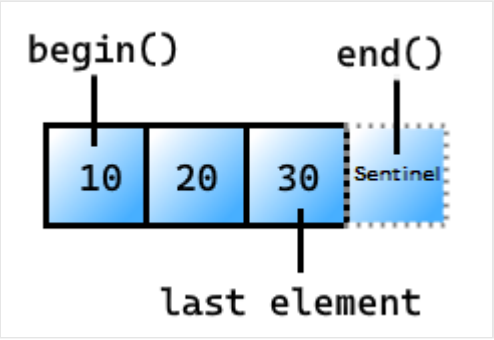
```
1) constexpr auto end() requires (!Simple_view<V> &&
!ranges::common_range<V>);
2) constexpr auto end() requires (!Simple_view<V> &&
ranges::common_range<V>);
3) constexpr auto end() const requires ranges::range<const V>;
4) constexpr auto end() const requires ranges::common_range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `keys_view`:



size

Get the number of elements.

C++

```
constexpr auto size() requires sized_range<V>
constexpr auto size() const requires sized_range<const V>
```

Parameters

None.

Return value

The number of elements in the `keys_view`.

Remarks

The size of the view is only available if the underlying range is a sized_range, or in other words, bounded.

See also

elements_view
values_view
<ranges>
View classes

`lazy_split_view` class (C++ Standard Library)

Article • 02/07/2023

Splits a range into subranges based on a delimiter. The delimiter can be a single element or a view of elements. The delimiter isn't part of the resulting subranges.

The primary differences between a split_view and a `lazy_split_view` are:

View	Can split a `const` range	range type
`split_view`	no	Supports forward_range or higher.
`lazy_split_view`	yes	Supports input_range or higher.

What makes a `lazy_split_view` "lazy" is that it doesn't lookahead for the next delimiter. That means it can support input_range whereas `split_view` requires at least `forward_range`. This is because `input_range` is single-pass whereas `forward_range` allows multi-pass iteration.

Prefer `split_view` because it's more efficient--unless you must split a range that is `const`. Regarding performance, `split_view` is more efficient.

A `lazy_split_view` has less efficient iterator increment and comparison than `split_view`, but is still O(1). A `split_view` has better performance when the distance between delimiters is small enough for subranges to fit in the CPU cache, in which case the delimiter lookahead effectively pre-caches the next subrange.

Syntax

C++

```
template<input_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
    indirectly_comparable<iterator_t<V>, iterator_t<Pattern>,
ranges::equal_to> &&
    (forward_range<V> || tiny_range<Pattern>)
class lazy_split_view : public view_interface<lazy_split_view<V, Pattern>>
```

Template parameters

`Pattern`

The type of the view that specifies a delimiter sequence.
The `(forward_range<V> || tiny-range <Pattern>)` requirement means that when the underlying range isn't forward_range, the delimiter must be a `tiny_range`. A `tiny_range` is a range with static extent whose size is 0 or 1. `tiny_range<T>` requires `sized_range<T>`, and `T::size()` must be a constant-expression that's less than or equal to 1.

`V`

The type of the underlying view.

Characteristics

For a description of the following entries, see View class characteristics

Range adaptor: lazy_split **Underlying range:** must satisfy input_range or higher **View iterator category:** same as the underlying range **Element type:** collection of `range_reference_t<V>` **Sized:** no **Common range:** Yes when the underlying range is both forward_range and `common`. **Borrowed range:** no **Is `const`-iterable:** only if the underlying range satisfies `forward_range` and is `const`-iterable

Members

Member functions	Description
Constructors	Construct the view.
base C++20	Get the underlying range.
begin C++20	Get an iterator to the first element in the view.
end C++20	Get the sentinel at the end of the view.
Inherited from view_interface	**Description**
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator bool C++20	Test whether the view isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `lazy_split_view`

```
C++

1) lazy_split_view() = default;
2) constexpr lazy_split_view(V base, Pattern pattern);
3) template<input_range R> requires constructible_from<V, views::all_t<R>>
&&
        constructible_from<Pattern, single_view<range_value_t<R>>>
        constexpr lazy_split_view(R&& rg, range_value_t<R> e);
```

Template parameters

`Pattern`

The type of the delimiter.

`R`

The type of the range.

`V`

The type of the underlying view.

Parameters

`e` A single element that identifies where to split the view. The element isn't part of the resulting subranges.

`base`

The underlying view.

`pattern`

The view of elements that identifies where to split the view. The view of elements isn't part of the resulting subranges.

`rg`

The range to split.

Return value

A `lazy_split_view` instance that contains one or more `subrange`s.

Remarks

The best way to create a `lazy_split_view` is by using the views::lazy_split range adaptor. Range adaptors are the intended way to create view classes. The view types are only exposed in case you want to create your own custom view type.

1) Create a `lazy_split_view` that has no elements. The underlying view is default constructed. `base()` returns a copy of `V()`.
2) Create a `lazy_split_view` by splitting the view using a delimiter sequence.
3) Create a `lazy_split_view` by splitting the view using a delimiter element.

Example: `lazy_split_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> rg{ 1, 2, 3, 1, 2, 3, 4, 5, 6 };

    // pipe syntax using range adaptor
    for (const auto& subrange : rg | std::views::split(3))
    {
        // Outputs:
        // 1 2
        // 1 2
        // 4 5 6
        for (const auto& elem : subrange)
        {
            std::cout << elem << ' ';
        }
        std::cout << '\n';
    }

    int delimiters[] = {2, 3};
    for (auto splitRange : std::views::split(rg, delimiters)) // ctor syntax
    {
        // outputs 1 1 4 5 6
        for (auto& i : splitRange)
        {
```

```
            std::cout << i << " ";
        }
    }
}
```

Output

```
1 2
1 2
4 5 6
1 1 4 5 6
```

base

Gets a copy of the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
1) constexpr V base() const & requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
2) constexpr V base() &&;
```

Parameters

None.

Returns

The underlying view.

begin

Get an iterator to the first element in the view.

C++

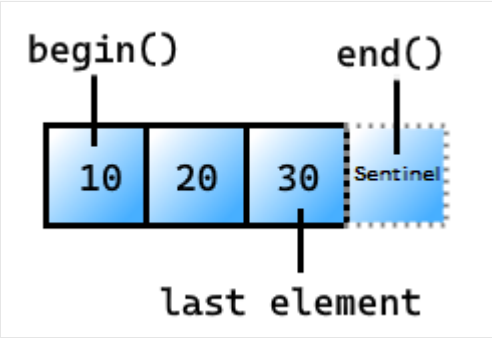
```
constexpr auto begin();
constexpr auto begin() const requires forward_range<V> &&
forward_range<const V>
```

Parameters

None.

Return value

An iterator pointing at the first element in the view.



end

Get the sentinel at the end of the view.

```
C++
```

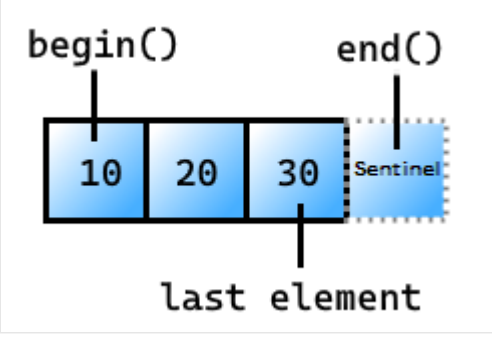
```
1) constexpr auto end() const;
2) constexpr auto end() requires forward_range<V> && common_range<V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the view:



Remarks

2) The `forward_range<V>` requirement means that the view `V` has at least a forward iterator. For more information about range iterators, see View class characteristics. The `common_range<V>` requirement means that the view `V` has identical iterator and sentinel types.

See also

<ranges>
lazy_split range adaptor split_view class
view classes

`owning_view` class (C++ Standard Library)

Article • 02/07/2023

A view that takes ownership of the elements in another range.

Syntax

C++

```
template<range R>
    requires std::movable<R> && (!is-initializer-list<R>)
class owning_view : public ranges::view_interface<owning_view<R>>;
```

Template parameters

R

The type of the underlying range.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::all
Underlying range	Must satisfy input_range or higher
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range satisfies `const-iterable`
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors	Construct an `owning_view`.
base[C++20]	Get a reference to the owned range.
begin[C++20]	Get an iterator to the first element.
data[C++20]	Get a pointer to the first element.
empty[C++20]	Test whether the view is empty.
end[C++20]	Get the sentinel at the end of the view.
size[C++20]	Get the number of elements.
operator=	Assign (move) the contents from another `owning_view` to this one.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
front[C++20]	Get the first element.
operator[][C++20]	Get the element at the specified position.
operator bool[C++20]	Test whether the view isn't empty.

Remarks

The best way to create an `owning_view` is by using the views::all range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

Even though this class owns its elements, it's not expensive to construct because the underlying range is moved using `std::move()`.

This view is useful when you want a range that doesn't depend on the lifetime of the container providing the elements.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of a `owning_view`.

```
C++

1) owning_view() requires default_initializable<R> = default;
2) constexpr owning_view(R&& rg);
3) owning_view(const owning_view& v) = delete; // no copy constructor
4) owning_view(const owning_view&& v) = default; // move constructor
```

Parameters

rg

The range to move to the `owning_view`.

v

The `owning_view` to move to the new `owning_view`.

For information about template parameter types, see Template parameters.

Remarks

1) The default constructor creates a default-initialized `owning_view`.
2) Move constructs the `owning_view` from *rg*.
3) An `owning_view` can't be copied, only moved.
4) Construct the `owning_view` from another `owning_view`.

Example: `owning_view`

```
C++

// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
#include <utility>

int main()
```

```
{
    std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};
    auto myOwningView = std::views::all(std::move(v)); // create an
owning_view from a moved vector
    std::cout << v.size() << '\n'; // outputs 0 because myOwningView now
owns the elements
    std::cout << myOwningView.size() << '\n'; // outputs 10

    std::vector<int> v2 = {1,2,3,4,5};
    std::ranges::owning_view<std::vector<int>> ov2{std::move(v2)};
    std::cout << v2.size() << '\n'; // outputs 0 because ov2 now owns the
elements
    std::cout << ov2.size() << '\n'; // outputs 5
}
```

Output

```
0
10
0
5
```

base

Gets a reference to the underlying range.

C++

```
1) constexpr R& base() & noexcept { return r_; }
2) constexpr const R& base() const & noexcept { return r_; }
3) constexpr R&& base() && noexcept { return std::move(r_); }
4) constexpr const R&& base() const && noexcept { return std::move(r_); }
```

Parameters

None.

Return value

A reference to the underlying range, call it *rg*.

For 1 & 2, the underlying range is returned via `return rg;`

For 3 & 4, the underlying range is returned via `std::move(rg);`

`begin`

Get an iterator to the first element in the view.

```
C++
```

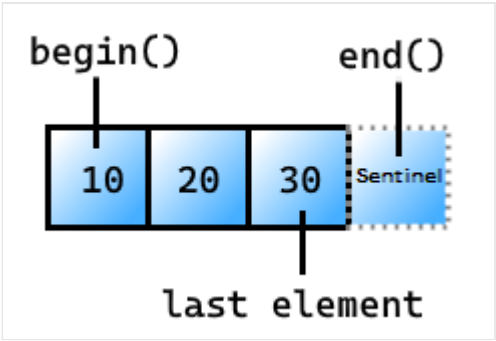
```
constexpr iterator_t<R> begin();
constexpr auto begin() const requires range<const R>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the view:



`data`

Get a pointer to the first element in the view.

```
C++
```

```
constexpr auto data()
    requires std::contiguous_iterator<ranges::iterator_t<R>>;
constexpr auto data() const
    requires std::contiguous_iterator<ranges::iterator_t<const R>>;
```

Parameters

None.

Return value

A pointer to the first element in the view.

Remarks

The underlying owned range must satisfy `contiguous_range`.

`empty`

Test whether the view is empty.

```
C++
```

```
constexpr bool empty();
constexpr bool empty() const;
```

Parameters

None.

Return value

Returns `true` if the underlying range has no elements. Otherwise, returns `false`.

`end`

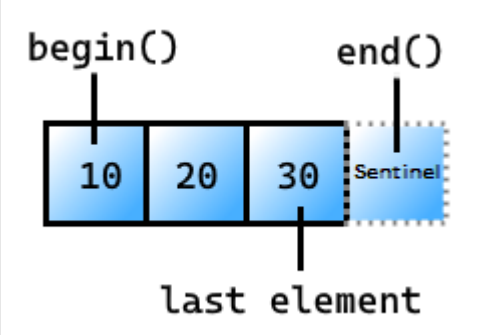
Get the sentinel at the end of the view.

```
C++
```

```
constexpr sentinel_t<R> end();
constexpr auto end() const requires range<const R>
```

Return value

The sentinel that follows the last element in the view:



`size`

Get the number of elements in the view.

```
C++
```

```
constexpr auto size() requires ranges::sized_range<R>;
constexpr auto size() const requires ranges::sized_range<const R>;
```

Parameters

None.

Return value

The number of elements in the view.

`operator=`

Assign (move) the contents from another `owning_view` to this one.

```
C++
```

```
owning_view& operator=(owning_view&&) = default;
```

Parameters

The `owning_view` to assign (move) to this one.

Return value

`*this`

Remarks

An `owning_view` can't be copied, only moved.

Example: `operator=`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v1 = {1,2,3};
    std::ranges::owning_view<std::vector<int>> ov1{std::move(v1)};

    std::vector<int> v2 = {4,5,6};
    std::ranges::owning_view<std::vector<int>> ov2{std::move(v2)};

    // operator=
    ov2 = std::move(ov1);

    // ov1 took ownership of v1, so v1 is empty
    // ov2 took ownership of v2, so v2 is empty
    // ov2 then took ownership of ov1, so ov1 is empty
    // ov2 now owns the elements 1, 2, 3

    std::cout << std::boolalpha << "v1.empty():" << v1.empty() << "
ov1.empty():" << ov1.empty() << '\n'; // v1.empty():true ov1.empty():true
    std::cout << "v2.empty():" << v2.empty() << " ov2.size():" << ov2.size()
<< '\n'; // v2.empty():true ov2.size():3

    for (auto e : ov2)
    {
         std::cout << e << ' '; // 1 2 3
    }
}
```

Output

```
v1.empty():true ov1.empty():true
v2.empty():true ov2.size():3
1 2 3
```

See also

<ranges>
all range adaptor
view classes

`ref_view` class (C++ Standard Library)

Article • 02/07/2023

A view that references the elements that belong to another range.

Syntax

C++

```
template<std::ranges::range R>
    requires std::is_object_v<R>
class ref_view : public ranges::view_interface<ref_view<R>>;
```

Template parameters

R

The range to reference.

Members

Member functions	Description
Constructors[C++20]	Construct a `ref_view`.
base[C++20]	Get a reference to the underlying range.
begin[C++20]	Get an iterator to the first element.
data[C++20]	Get a pointer to the first element in the referenced range.
empty[C++20]	Test whether this `ref_view` is empty.
end[C++20]	Get the sentinel at the end of this `ref_view`.
size[C++20]	Get the number of elements. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
front[C++20]	Get the first element.

Member functions	Description
operator[]C++20	Get the element at the specified position.
operator boolC++20	Test whether this `ref_view` isn't empty.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::all or views::common
Underlying range	Must satisfy input_range
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Yes
Common range	Only if the underlying range satisfies common_range
Borrowed range	Yes

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `ref_view`

C++

```
// construct a ref_view from a range
template<different-from<ref_view> R>
constexpr ref_view(R&& rg);
```

Parameters

rg
The range to reference.

For information about the template parameter type, see Template parameters.

Return value

A `ref_view` instance.

Remarks

The best way to create a `ref_view` is by using the views::all range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

A `ref_view` is useful for converting a container to a view. For example, you can use `ref_view` to convert a `vector` to a view, which makes it inexpensive to pass the elements of the vector around.

Example: `ref_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {1,2,3};
    auto refView = std::views::all(v);
    std::cout << &refView[1] << " : " << &v[1]; // outputs two identical
memory addresses, e.g. 00000239AFAFDF90 : 00000239AFAFDF90
    refView[0] = 10; // modifies v[0]
    std::cout << "\n" << v[0]; // 10
}
```

Output

```
00000239AFAFDF90 : 00000239AFAFDF90
10
```

`base`

Gets a copy of the underlying range.

```
C++
```

```
constexpr R& base() const;
```

Parameters

None.

Return value

The underlying range.

`begin`

Get an iterator to the first element in the `ref_view`.

```
C++
```

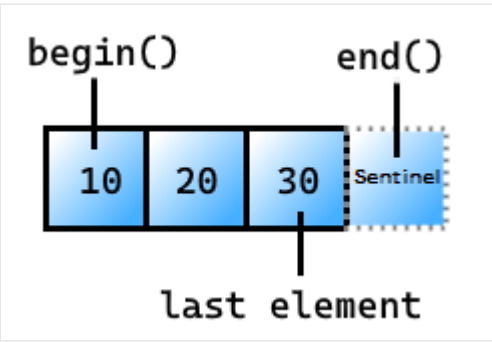
```
constexpr iterator_t<R> begin() const;
```

Parameters

None.

Return value

An iterator pointing at the first element in this `ref_view`.



`data`

Get a pointer to the first element in this `ref_view`. The elements in the range must be contiguous.

```
C++
```

```
constexpr auto data() const requires contiguous_range<R>;
```

Parameters

None.

Return value

A pointer to the first element.

`empty`

Test whether this `ref_view` is empty.

```
C++
```

```
constexpr bool empty() const
```

Parameters

None.

Return value

Returns `true` if the `ref_view` contains no elements. Otherwise, `false`.

`end`

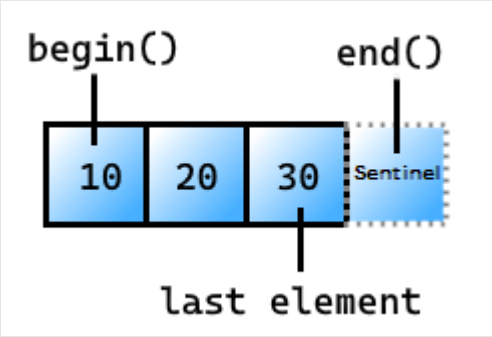
Get the sentinel at the end of this `ref_view`.

```
C++
```

```
constexpr sentinel_t<R> end() const
```

Return value

The sentinel that follows the last element in this `ref_view`:



`size`

Get the number of elements.

```
C++
```

```
constexpr auto size() const requires sized_range<R>
```

Parameters

None.

Return value

The number of elements in the `ref_view`.

See also

<ranges>
all range adaptor
view classes

`reverse_view` class (C++ Standard Library)

Article • 02/07/2023

A view of the elements of a range in reverse order.

Syntax

```
C++
```

```
template<ranges::view V>
requires ranges::bidirectional_range<V>
class reverse_view : public ranges::view_interface<reverse_view<V>>;
```

Template parameters

V

The type of the underlying view.
This type must satisfy `ranges::bidirectional_range`.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::reverse
Underlying range	Must satisfy bidirectional_range up to random_access_range
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range is a common_view and satisfies `const-iterable`
Common range	Yes

Characteristic	Description
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors[C++20]	Construct a `reverse_view`.
base[C++20]	Get the underlying range.
begin[C++20]	Get an iterator to the first element.
end[C++20]	Get the sentinel at the end of the `reverse_view`.
size[C++20]	Get the number of elements.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
empty[C++20]	Test whether the `reverse_view` is empty.
front[C++20]	Get the first element.
operator[][C++20]	Get the element at the specified position.
operator bool[C++20]	Test whether the `reverse_view` isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `reverse_view` from a bidirectional view.

C++

```
1) reverse_view() requires default_initializable<V> = default; // default-
constructs the underlying view
```

```
2) constexpr explicit reverse_view(V rg); // initializes the underlying view
via std::move(r)
```

Parameters

rg

The view to provide a reversed view of.

For information about the template parameter type, see Template parameters.

Return value

A view of the underlying range, in reverse order.

Remarks

The best way to create a `reverse_view` is by using the views::reverse range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) The default constructor default-initializes a `reverse_view`.
2) Create a `reverse_view` from the specified view.

Example: `reverse_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};
    auto rv = v | std::views::reverse;

    for (auto e : rv) // 6 5 -4 3 2 1 0
    {
        std::cout << e << ' ';
    }
}
```

Output

```
6 5 -4 3 2 1 0
```

base

Gets a copy of the underlying view.

```
C++
```

```
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
constexpr V base() &&;
```

Parameters

None.

Return value

The underlying view.

begin

Get an iterator to the first element in the `reverse_view`.

```
C++
```

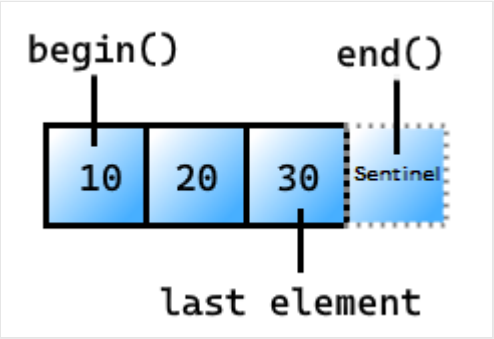
```
1) constexpr reverse_iterator<iterator_t<V>> begin();
2) constexpr reverse_iterator<iterator_t<V>> begin() requires
common_range<V>;
3) constexpr auto begin() const requires common_range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the `reverse_view`.



Remarks

After the first call to `begin()`, subsequent calls run in constant time, O(1), no matter how many elements are in the `reverse_view`. This has implications because `reverse_view` caches the value of `last` so it can return it repeatedly from `begin`. This means you shouldn't reuse a view after the underlying container is modified. If the underlying range is modified, generate a new view, which is inexpensive.

2) The underlying view must satisfy common_range, which means that the underlying view must have the same begin and end iterator type.
3) The underlying view must satisfy `common_range` for a const view to iterate over a `const reverse_view`.

end

Get the sentinel at the end of the `reverse_view`

C++

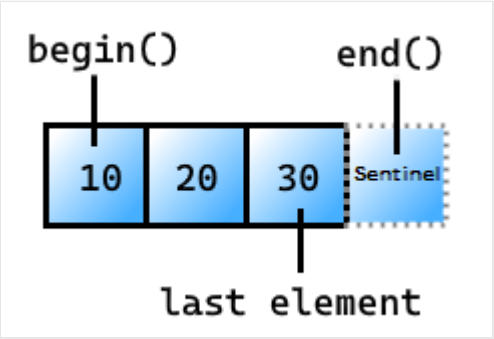
```
1) constexpr reverse_iterator<iterator_t<V>> end();
2) constexpr auto end() const requires common_range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `reverse_view`.



Remarks

For 2, the underlying view must satisfy common_range for a const view, which means that the underlying view must have the same begin and end iterator type.

`size`

Get the number of elements.

C++

```
constexpr auto size() requires ranges::sized_range<V>;
constexpr auto size() const requires ranges::sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the `reverse_view`.

Remarks

The size of the view is only available if the underlying range is a sized_range, or in other words, bounded.

See also

<ranges>
reverse range adaptor
View classes

`single_view` class (C++ Standard Library)

Article • 02/07/2023

A view that has only one element. This view is useful for test purposes for calling code that needs to be provided with a view with at least one element in it.

Syntax

```
C++
```

```
template<std::copy_constructible T>
  requires std::is_object_v<T>
class single_view : public ranges::view_interface<single_view<T>>
```

Template parameters

`T`

The type of the element.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::single
Underlying range	None
Element type	Specified when the `single_view` is created
View iterator category	`contiguous_range`
Sized	Always returns 1
Is `const`-iterable	Yes
Common range	Yes
Borrowed range	No

Members

Member functions	Description
Constructors C++20	Construct a `single_view`.
begin C++20	Get an iterator to the element.
data C++20	Get a pointer to the element.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements. Always returns `1`.
Inherited from view_interface	**Description**
back C++20	Get the element.
empty C++20	Test whether the view is empty (always returns `false`).
front C++20	Get the element.
operator[] C++20	Get the element at the specified position (only position 0 is valid).
operator bool C++20	Test whether the view isn't empty (always returns `false`).

Remarks

The best way to create a `single_view` is by using the views::single range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

The value in the `single_view` can be modified unless the template value is `const`. For example: `single_view<const float> sv{3.14} // this value can't be modified because it's const`.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create an instance of a `single_view`.

```
C++
```

```
1) single_view() = default;
2) constexpr explicit single_view(const T& t);
3) constexpr explicit single_view(T&& t);
4) template<class... Args>
        requires constructible_from<T, Args...>
    constexpr single_view(in_place_t, Args&&... args);
```

Parameters

t
The element value.

For information about the template parameter type, see Template parameters.

Remarks

The best way to create a `single_view` is by using the views::single range adaptor.

1) Create a `single_view` with a single element of the specified type that is default constructed. For example, `single_view<float> sv{}` creates a `single_view` with a single element of type `float` that is default constructed to `0.0`.
2) Create a `single_view` with a single element of the specified type that is copy-initialized from the specified argument. For example, `single_view<myObjectType> sv{myObject}` creates a `single_view` with a single element of type `myObjectType` that is copy-initialized from the argument.
3) Create a `single_view` with a single element of the specified type that is move-initialized from the argument.
4) Create a `single_view` with a single element of the specified type initialized with `(std::forward<Args>(args)...)`.

Example `single_view`

```
C++
```

```
/// requires /std:c++20 or higher
#include <ranges>
```

```cpp
#include <iostream>
#include <string>
#include <tuple>

int main()
{
    std::ranges::single_view<int> sv{7};
    std::cout << sv.front() << " " << *sv.data() << "\n"; // 7 7

    std::ranges::single_view<std::tuple<int, std::string>> sv2{{6502, "8-
bit"}};
    std::cout << std::get<0>(sv2[0]) << " " << std::get<1>(sv2[0]) << "\n";
// 6502 8-bit
}
```

Output

```
7 7
6502 8-bit
```

`begin`

Get a pointer to the single element in the view.

C++

```cpp
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
```

Parameters

None.

Return value

A pointer to the single element inside the `single_view`.

`data`

Get a pointer to the single element in the `single_view`.

C++

```
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

Parameters

None.

Return value

A pointer to the element in the `single_view`.

`end`

Gets a pointer to the sentinel after the element.

C++

```
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
```

Parameters

None.

Return value

A pointer to the sentinel that follows the element.

`size`

Get the number of elements in the view. Always returns `1`.

C++

```
static constexpr size_t size() noexcept;
```

Parameters

None.

Return value

`1`

See also

<ranges>
single range adaptor
empty_view
View classes

`split_view` class (C++ Standard Library)

Article • 02/07/2023

Splits a view into subranges based on a delimiter. The delimiter can be a single element or a view of elements. The delimiter isn't part of the resulting `split_view`.

A related view is the lazy_split_view class. The primary differences between `split_view` and `lazy_split_view` are:

View	Can split a `const` range	range type
`split_view`	no	Supports forward_range or higher.
`lazy_split_view`	yes	Supports input_range or higher.

Prefer `split_view` because it's more efficient unless you must split a range that is `const`.

Syntax

```
C++
```

```
template<forward_range V, forward_range Pattern>
    requires view<V> && view<Pattern> &&
    indirectly_comparable<iterator_t<V>, iterator_t<Pattern>,
ranges::equal_to>
class split_view : public view_interface<split_view<V, Pattern>>;
```

Template parameters

`Pattern`
The type of the view that specifies the delimiter sequence.

`V`
The type of the underlying view.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description

Characteristic	Description
Range adaptor	views::split
Underlying range	Same as underlying range
Element type	`range_reference_t<V>`
View iterator category	Satisfies forward_range
Sized	No
Is `const`-iterable	No
Common range	Only if the underlying range satisfies common_range
Borrowed range	No

Members

Member functions	Description
Constructors	Construct the view.
base[C++20]	Get the underlying range.
begin[C++20]	Get an iterator to the first element.
end[C++20]	Get the sentinel at the end of the view.
Inherited from view_interface	**Description**
empty[C++20]	Test whether the view is empty.
front[C++20]	Get the first element.
operator bool[C++20]	Test whether the view isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `split_view`

```
C++
```

```
1) split_view() requires default_initializable<V> &&
default_initializable<Pattern> = default;
2) constexpr split_view(V base, Pattern pattern);
3) template<input_range R> requires constructible_from<V, views::all_t<R>>
&&
     constructible_from<Pattern, single_view<range_value_t<R>>>
     constexpr split_view(R&& rg, range_value_t<R> e);
```

Parameters

e

A single element that identifies where to split the view. The element isn't part of the resulting view.

base

The underlying view.

pattern

The view of elements that identifies where to split the view. The view of elements isn't part of the resulting view.

rg

The range to split.

For information about template parameter types, see Template parameters.

Return value

A `split_view` instance that contains one or more subranges.

Remarks

The best way to create a `split_view` is by using the views::split range adaptor. Range adaptors are the intended way to create view classes. The view types are only exposed in case you want to create your own custom view type.

1) Create a `split_view` that is default constructed. The underlying view and `pattern` are default constructed. `base()` returns a copy of `V()`.

2) Create a `split_view` by splitting the view using a sequence of delimiters.

3) Create a `split_view` by splitting the view using a single delimiter.

Example `split_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> rg{ 1, 2, 3, 1, 2, 3, 4, 5, 6 };

    // pipe syntax using range adaptor
    for (const auto& subrange : rg | std::views::split(3))
    {
        // outputs
        // 1 2
        // 1 2
        // 4 5 6
        for (const auto& elem : subrange)
        {
            std::cout << elem << ' ';
        }
        std::cout << '\n';
    }

    int delimiters[] = {2, 3};
    for (auto splitRange : std::views::split(rg, delimiters)) // ctor syntax
    {
        for (auto& i : splitRange)
        {
            std::cout << i << " "; // 1 1 4 5 6
        }
    }
}
```

Output

```
1 2
1 2
4 5 6
1 1 4 5 6
```

`base`

Gets a copy of the underlying view.

```
C++
```

```
// Uses a copy constructor to return the underlying view
1) constexpr V base() const & requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
2) constexpr V base() &&;
```

Parameters

None.

Returns

The underlying view.

`begin`

Get an iterator to the first element in the view.

```
C++
```

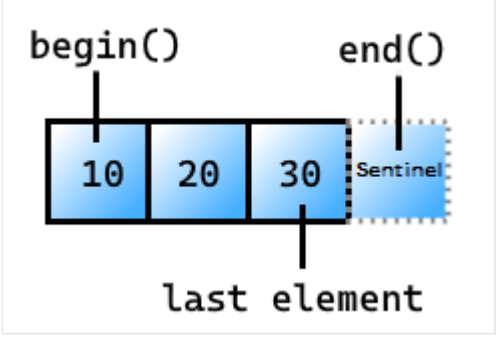
```
constexpr auto begin();
```

Parameters

None.

Return value

An iterator pointing at the first element in the view.



end

Get the sentinel at the end of the view.

```
C++
```

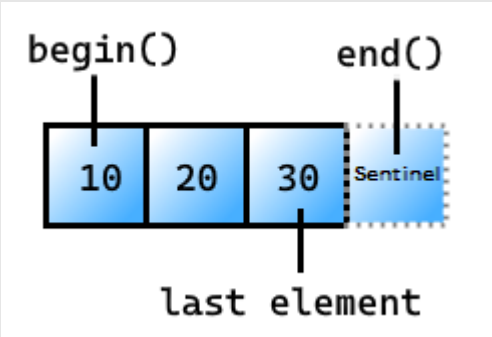
```
constexpr auto end();
```

Parameters

None.

Return value

The sentinel that follows the last element in the view:



See also

<ranges>
split_view range adaptor
lazy_split_view class
view classes

`subrange` class (C++ Standard Library)

Article • 02/07/2023

Provides a view of part of the elements of a range as defined by a begin iterator and sentinel.

Syntax

```
C++
```

```
template<input_or_output_iterator I, sentinel_for<I> S, subrange_kind K>
  requires (K == subrange_kind::sized || !sized_sentinel_for<S, I>)
class subrange : public view_interface<subrange<I, S, K>>
```

Template parameters

`I`

The begin iterator type. The input_or_output_iterator concept ensures that *`I`* is an iterator that can read all of the elements.

`K`

The kind of subrange: Use `subrange_kind::sized` to specify a sized subrange. Use `sized_sentinel_for<S, I>` if the iterator and sentinel can be subtracted to yield the size. The requirement `subrange_kind::sized || !sized_sentinel_for<S, I>` stores the size locally in the subrange object, and requires that you construct the subrange either using the constructor that takes a sized_range (for which you would specify `subrange_kind::sized` here) or via the constructor that takes an `iterator`, `sentinel`, and `size` (so you would specify `sized_sentinel_for<S, I>` here).

`S`

The end iterator type. The sized_sentinel_for concept ensures that *`S`* can be used as a sentinel for *`I`* and that it's possible to compute the distance between the sentinel and the current iterator position in *`I`* in constant time.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description

Characteristic	Description
Range adaptor	views::counted
Underlying range	Any range
Element type	`iter_reference_t<I>`
View iterator category	Same as *I*s category
Sized	If *K* is `subrange::sized`
Is `const`-iterable	If *I* is copyable
Common range	If *I* and *S* are the same type.
Borrowed range	Yes

Members

Member functions	Description
Constructors[C++20]	Construct a `subrange`.
operator PairLike[C++20]	Convert a `subrange` to a pair-like type.
advance[C++20]	Move the iterator a specified distance.
begin	Get an iterator to the first element.
empty[C++20]	Test whether the `subrange` is empty.
end[C++20]	Get the sentinel at the end of the `subrange`.
next[C++20]	Creates a copy of this `subrange` but with the stored iterator moved forward the specified distance.
prev[C++20]	Creates a copy of this `subrange` but with the stored iterator moved back the specified distance.
size[C++20]	Get the number of elements.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
data[C++20]	Get a pointer to the first element.
front[C++20]	Get the first element.

Member functions	Description
operator[]C++20	Get the element at the specified position.
operator boolC++20	Test whether the `subrange` is empty.

Remarks

A `subrange` is useful when you have a begin and end iterator, but you want to pass a single object instead. For example, if you wanted to call a range adaptor but had begin and end iterators, you could use a `subrange` to wrap them and pass the `subrange` to the range adaptor.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Create a `subrange`.

C++

```
1) subrange() requires default_initializable<I> = default;
2) template <Convertible_to_non_slicing<I> It>
    constexpr subrange(It begin, S end) requires (!Store_size);
3) template <Convertible_to_non_slicing<I> It>
    constexpr subrange(It begin, S end, const Size_type size) requires (K ==
subrange_kind::sized);
4) template <Not_same_as<subrange> rg>
    requires borrowed_range<rg>
        && Convertible_to_non_slicing<iterator_t<rg>, I>
        && convertible_to<sentinel_t<rg>, S>
    constexpr subrange(rg&& r) requires (!_Store_size || sized_range<rg>);
5) template <borrowed_range rg>
        requires Convertible_to_non_slicing<iterator_t<rg>, I> &&
convertible_to<sentinel_t<rg>, S>
    constexpr subrange(rg&& r, const _Size_type sizeHint) requires (K ==
subrange_kind::sized)
```

Parameters

`begin`
Iterator that points to the first element in the subrange.

`end`
Sentinel that points to the end of the subrange. The element it points to isn't included in the subrange.

`sizeHint`
The size of the range in elements. This is used to optimize the `size` member function and is necessary if you want to make a sized `subrange` from an iterator and sentinel whose types don't model sized_sentinel_for.

For information about template parameter types, see Template parameters.

Return value

A `subrange` instance.

Remarks

1) Default constructs the stored iterator and sentinel. The size hint is set to 0.
2) Uses `std::move()` to move the `begin` iterator and `end` sentinel to the stored iterator and sentinel.
3) Initializes the stored iterator with `std::move(begin)`, the stored sentinel with `std::move(end)`, and the stored size hint with `size`, which should equal the distance between the first and second arguments.
4) Construct a `subrange` from a range.
5) The behavior isn't defined if `szHint != ranges::distance(rg)`.

The counted range adaptor can create a `subrange`. That adaptor takes a begin iterator and a count.

Example: `counted`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
```

```
int main()
{
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto pos5 = std::ranges::find(v, 5);
    auto countedView = std::views::counted(pos5, 5);
    for (auto e : countedView) // outputs 5 6 7 8 9
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';

    // You can pass the range directly if it supports
input_or_output_iterator, in which case, the
    // count starts from the first element
    const char chars[] = { 'H','i',' ','t','h','e','r','e' };
    for (char c : std::views::counted(chars, 2))
    {
        std::cout << c; // outputs Hi
    }
}
```

Output

```
5 6 7 8 9
Hi
```

`operator PairLike`

Convert a `subrange` to a type that models `pair-like`.

C++

```
template<not-same-as<subrange> PairLike>
requires pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

Parameters

None.

For information about template parameter types, see Template parameters.

Return value

A `PairLike` value that is direct-initialized with the stored iterator and sentinel. The last value in the pair will be the sentinel.

Remember that the sentinel is *past* the last element in the subrange, as shown in the example below.

Remarks

This conversion is useful with older `Boost::Ranges` code that accepts (first, last) pairs to denote a range.
This conversion is useful for converting a subrange to a `pair` or `tuple` or other type that models `pair_like`. Some examples of `pair_like` types are:

```
C++
```

```
std::array<T, 2>
std::pair<T, U>
std::ranges::subrange<I, S, K>
std::tuple<T, U>
```

Example: `operator PairLike()`

```
C++
```

```
// requires /std:c++20 or later
#include <iostream>
#include <ranges>
#include <vector>
#include <utility>

int main()
{
    constexpr int a[] = {0, 1, 2, 3, 4, 5};
    std::ranges::subrange rg(a);
    rg.advance(2);
    const std::pair<const int*, const int*> p = rg;
    for (auto e : rg)
    {
        std::cout << e << ' ';
    }

    // because the sentinel points after the last element, subtract one to
get the last element
    std::cout << '\n' << *p.first << ':' << *(p.second - 1) << '\n'; //
outputs 2:5
 }
```

Output

```
2 3 4 5
2:5
```

`advance`

Adjust the iterator for this `subrange` by `n` elements.

C++

```
constexpr subrange& advance(const iter_difference_t<I> n);
```

Parameters

`n`
How many elements to adjust the iterator. `n` can be positive (move forward) or, if `I` is bidirectional, negative (move backward).

Remarks

This function modifies the current state of the iterator in the `subrange`.

If you advance past the end of the `subrange`, the iterator is set to the sentinel at the end of the `subrange`.
If you advance past the beginning of the `subrange` (using a negative `n`), you'll get an invalid parameter exception if the range you made the `subrange` from doesn't have an element in the place.

Example `advance`

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <ranges>
#include <string>
#include <vector>

void print(const std::string &msg, auto &&v)
{
    std::cout << msg << '\n';
```

```
    for (auto& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    print("Original vector: ", v); // outputs 0 1 2 3 4 5 6 7 8 9 10

    // create a subrange 3 4 5 6
    std::ranges::subrange theSubrange{ std::ranges::find(v,3),
std::ranges::find(v, 7) };
    print("The subrange: ", theSubrange); // outputs 3 4 5 6

    auto sr = theSubrange.advance(2); // get a subrange 2 positions to the
right of the current iterator location
    print("theSubrange.advance(2): ", sr); // outputs 5 6
    print("Note that subrange's iterator moved during advance(): ", sr); //
outputs 5 6
    sr = theSubrange.advance(-3); // Moving before the subrange, but onto a
valid element in the original range
    print("theSubrange.advance(-3): ", sr); // outputs 2 3 4 5 6
}
```

Output

```
Original vector:
0 1 2 3 4 5 6 7 8 9 10
The subrange:
3 4 5 6
theSubrange.advance(2):
5 6
Note that subrange's iterator moved during advance():
5 6
theSubrange.advance(-3):
2 3 4 5 6
```

begin

Get an iterator to the first element in the `subrange`.

C++

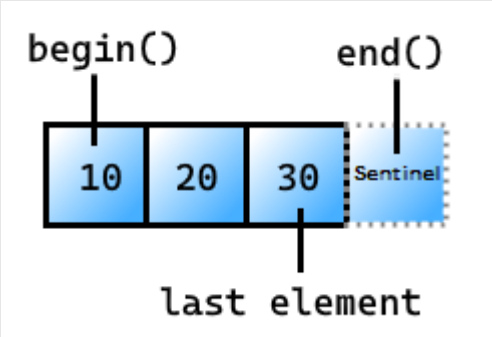
```
1) constexpr I begin() const requires copyable<I>;
2) [[nodiscard]] constexpr I begin() requires (!std::copyable<I>);
```

Parameters

None.

Return value

An iterator pointing at the first element in the `subrange`. If the iterator isn't copyable, it's returned with `std::move()`. If the iterator is moved, the state of the stored iterator depends on the implementation of the move constructor for `I`.



`empty`

Test whether the `subrange` is empty.

C++

```
constexpr bool empty() const;
```

Parameters

None.

Return value

Returns `true` if the `subrange` has no elements. Otherwise, returns `false`.

`end`

Get the sentinel at the end of the `subrange`

C++

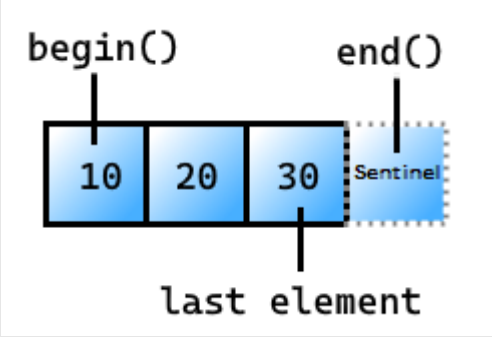
```
[[NODISCARD]] constexpr S end() const;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `subrange`:



The sentinel is copy-constructed from the stored sentinel.

`next`

Creates a copy of this `subrange` but with the stored iterator moved forward the specified distance.

```
C++

1) [[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const &
requires forward_iterator<I>;
2) [[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;
```

Parameters

n

How many elements to move the iterator forward. Defaults to 1. Must be positive.

Return value

Returns a copy of the `subrange` starting at the *`n`*th element.

Remarks

Unlike `advance()`, `next()` doesn't change the location of the iterator stored in the original `subrange`. The returned `subrange` has all the elements that the original subrange has, but the iterator is in a different location.

1) The return value is the same as:

C++

```
auto tmp = *this;
tmp.advance(n);
return tmp;
```

2) The return value is the same as:

C++

```
advance(n);
return std::move(*this);
```

Example: `next`

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <ranges>
#include <string>
#include <vector>

void print(const std::string &msg, auto &&v)
{
    std::cout << msg << '\n';
    for (auto& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    print("Original vector:", v); // 0 1 2 3 4 5 6 7 8 9 10

    // create a subrange from the front of v up to (but not including) the
element 7
```

```
    std::ranges::subrange theSubrange{ std::ranges::find(v,1),
std::ranges::find(v, 7) };
    print("The subrange:", theSubrange); // 1 2 3 4 5 6

    auto forward = theSubrange.advance(3); // get a subrange 3 positions to
the right of the current iterator location
    print("theSubrange.advance(3):", forward); // 4 5 6

    // prev()
    auto previous = theSubrange.prev(2); // move back 2
    print("theSubrange.prev(2):", previous); // 2 3 4 5 6
    print("Note that the subrange's iterator did *not* move during prev():",
theSubrange); // 4 5 6
}
```

Output

```
Original vector:
0 1 2 3 4 5 6 7 8 9 10
The subrange:
1 2 3 4 5 6
theSubrange.next(3):
4 5 6
Note that the original subrange's iterator did *not* move during next():
1 2 3 4 5 6
```

prev

Creates a copy of this `subrange`, but with the stored iterator moved back the specified distance.

C++

```
[[nodiscard]] constexpr subrange prev(std::iter_difference_t<I> n = 1 )
const
    requires std::bidirectional_iterator<I>;
```

Parameters

n

How many elements to move the iterator back. Defaults to 1. Must be positive.

Return value

Returns a copy of the `subrange` but with the iterator moved back *n* elements.

Remarks

Unlike `advance()`, `prev()` doesn't change the location of the iterator stored in the original `subrange`.

The returned `subrange` has all the elements that the original subrange has, but the iterator is just in a different location. You can think of the return value as:

C++

```
auto tmp = *this;
tmp.advance(-n);
return tmp;
```

Example `prev`

C++

```
// requires /std:c++20 or later
#include <iostream>
#include <ranges>
#include <string>
#include <vector>

void print(const std::string &msg, auto &&v)
{
    std::cout << msg << '\n';
    for (auto& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    print("Original vector:", v); // 0 1 2 3 4 5 6 7 8 9 10

    // create a subrange from the front of v up to (but not including) the
element 7
    std::ranges::subrange theSubrange{std::ranges::find(v,1),
std::ranges::find(v, 7)};
    print("The subrange: ", theSubrange); // 1 2 3 4 5 6

    auto forward = theSubrange.advance(3); // get a subrange 3 positions to
the right of the current iterator location
    print("theSubrange.advance(3):", forward); // 4 5 6

    // prev()
```

```
    auto previous = theSubrange.prev(2); // move back 2
    print("theSubrange.prev(2):", previous); // 2 3 4 5 6
    print("Note that the subrange's iterator did *not* move during prev():",
theSubrange); // 4 5 6
}
```

Output

```
Original vector:
0 1 2 3 4 5 6 7 8 9 10
The subrange:
1 2 3 4 5 6
theSubrange.advance(3):
4 5 6
theSubrange.prev(2):
2 3 4 5 6
Note that the subrange's iterator did *not* move during prev():
4 5 6
```

`size`

Get the number of elements in the `subrange`.

C++

```
constexpr size() const
    requires (K == ranges::subrange_kind::sized);
```

Parameters

None.

Return value

The number of elements in the `subrange`.

If the size isn't stored, which is the case when the `subrange` is created with `K == ranges::subrange_kind::sized` specified and `std::sized_sentinel_for<S, I>` isn't satisfied, then the size is returned as the distance between the begin and end iterators.

Changing the position of the `begin` iterator, with `advance`, for example, changes the reported size.

See also

<ranges>
counted
View classes

`take_view` class (C++ Standard Library)

Article • 02/07/2023

A view of the first *N* elements from another view.

Syntax

C++

```
template<view V>
class take_view : public view_interface<take_view<V>>;
```

Template parameters

V

The type of the underlying range.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::take
Underlying range	Any range
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	No
Is `const`-iterable	Only if the underlying range is `const` iterable
Common range	Only if the underlying range satisfies random_access_range and sized_range
Borrowed range	Only if the underlying range is borrowed_range

Members

Member functions	Description
Constructors	Construct the view.
base C++20	Get the underlying range.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the view.
size C++20	Get the number of elements. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back C++20	Get the last element.
data C++20	Get a pointer to the first element.
empty C++20	Test whether the view is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the view isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `take_view`

C++

```
1) take_view() requires default_initializable<V> = default;
2) constexpr take_view(V base, range_difference_t<V> count);
```

Parameters

base

The underlying view.

count

The number of elements to take from the front of the underlying view. If *count* is more than the number of elements in the underlying view, the view will contain all the elements in the underlying range.

For information about the template parameter type, see Template parameters.

Return value

A `take_view`, which is a view of the first *N* elements from another view. If you specify more elements to drop than exist in the underlying range, an `empty_view` is returned.

Remarks

The best way to create a `take_view` is by using the views::take range adaptor. Range adaptors are the intended way to create view classes. The view types are only exposed in case you want to create your own custom view type.

1) Create a `take_view` that has no elements. The underlying view is default constructed. `base()` returns a copy of `V()`.
2) Create a `take_view` from a *base* and a count. *base* is moved via `std::move()`.

If *count* is less than the number of elements in the underlying range, then `count` determines the size of the `take_view`.
If *count* is greater than the number of elements in the underlying range, then the `take_view` includes all of the elements in the underlying range.

Example: `take_view`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6};
```

```
    auto newView = std::views::take(v, 3);

    for (auto& e : newView)
    {
        std::cout << e << ' '; // 0 1 2
    }
    std::cout << '\n';

    // Use the '|' operator to create a take_view
    for (auto i : v | std::views::take(3))
    {
        std::cout << i << ' '; // 0 1 2
    }
}
```

Output

```
0 1 2
0 1 2
```

base

Gets a copy of the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
1) constexpr V base() const & requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
2) constexpr V base() &&;
```

Parameters

None.

Return value

The underlying view.

begin

Get an iterator to the first element in the view.

C++

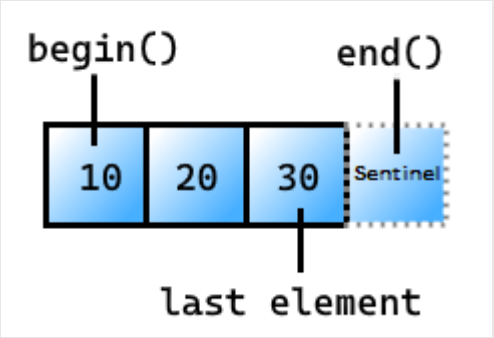
```
constexpr auto begin() requires (!Simple_view<V>);
constexpr auto begin() const requires range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the view.



Remarks

For 1, the `Simple_view<V>` requirement means that the view `V` and `const V` must have the same iterator and sentinel types.

end

Get the sentinel at the end of the view.

C++

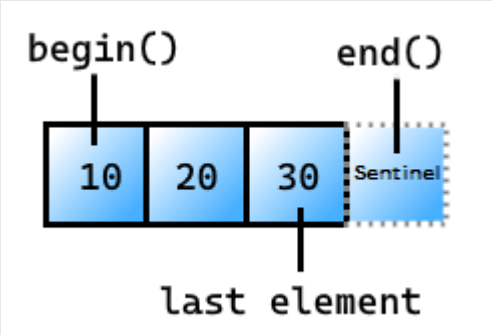
```
1) constexpr auto end() requires !(Simple_view<V>);
2) constexpr auto end() const requires range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the view.



Remarks

For 1, the `Simple_view<V>` requirement means that the view `V` and `const V` must have the same iterator and sentinel types.

`size`

Get the number of elements.

```
C++

constexpr auto size() requires sized_range<V>;
constexpr auto size() const requires sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the `take_view`.
If the `take_view` was constructed with an explicit *`count`*:

- if *`count`* is less than the number of elements in the underlying range, it's returned as the size of the view.
- if *`count`* is greater than the number of elements in the underlying range, then the size of the view is `ranges::size(base)`.

See also

<ranges>
take range adaptor
view classes

`take_while_view` class (C++ Standard Library)

Article • 02/07/2023

A view that contains the leading elements of a range that match a predicate.

Syntax

```
C++
```

```
template<view V, class Pred> requires
  input_range<V> && is_object_v<Pred> &&
  indirect_unary_predicate<const Pred, iterator_t<V>>
class take_while_view : public view_interface<take_while_view<V, Pred>>;
```

Template parameters

`Pred`

The type of the predicate that determines the leading elements to put in the view.

`V`

The type of the underlying view.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::take_while
Underlying range	Must satisfy input_range or higher
Element type	Same as the underlying range
View iterator category	Same as the underlying range
Sized	No
Is `const`-iterable	Only if the underlying range is `const` iterable and the predicate can work with `const` references.

Characteristic	Description
Common range	No
Borrowed range	No

Members

Member functions	Description
Constructors[C++20]	Construct the view.
base[C++20]	Get the underlying range.
begin[C++20]	Get an iterator to the first element.
end[C++20]	Get the sentinel at the end of the view.
pred[C++20]	Get a reference to the predicate that determines which elements to take.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
data[C++20]	Get a pointer to the first element.
empty[C++20]	Test whether the view is empty.
front[C++20]	Get the first element.
operator[][C++20]	Get the element at the specified position.
operator bool[C++20]	Test whether the view isn't empty.
size	Get the number of elements in the view.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `take_while_view`

```
C++

1) take_while_view() requires
    default_initializable<V> &&
    default_initializable<Pred> = default;

2) constexpr take_while_view(V base, Pred pred);
```

Parameters

`base`
The underlying view.

`pred`
The predicate that determines the leading elements to put in the view.

For information about template parameter types, see Template parameters.

Return value

A `take_while_view` object.

Remarks

The best way to create a `take_while_view` is by using the views::take_while range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Move constructs the `take_while_view` from a *`base`* view and a *`pred`* predicate. Both *`base`* and *`pred`* are moved via `std::move()`.
2) Constructs an empty `take_while_view`. The underlying view and predicate are default constructed.

Example: `take_while_view`

```
C++

// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
```

```
int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};
    auto twv = std::views::take_while(v, [](int i) {return i >= 0; });

    for (auto& e : twv)
    {
        std::cout << e << ' '; // 0 1 2 3
    }
    std::cout << '\n';

    // Using the '|' operator to create a take_view
    for (auto i : v | std::views::take_while([](int i) {return i < 5; }))
    {
        std::cout << i << ' '; // 0 1 2 3 -4
    }
}
```

Output

```
0 1 2 3
0 1 2 3 -4
```

base

Gets a copy of the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
1) constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
2) constexpr V base() &&;
```

Parameters

None.

Returns

A copy of the underlying view.

begin

Get an iterator to the first element in the view.

```
C++
```

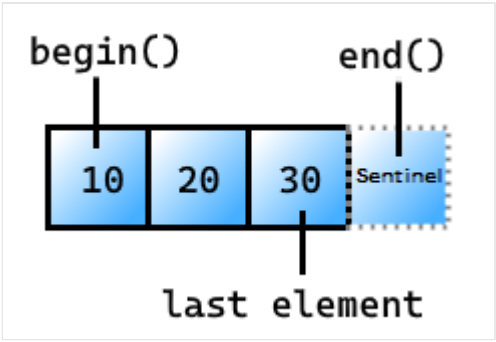
```
1) constexpr auto begin() requires (!Simple_view<V>);
2) constexpr auto begin() const requires
        range<const V> &&
        indirect_unary_predicate<const Pred, iterator_t<const V>>
```

Parameters

None.

Return value

An iterator pointing at the first element in the view. The behavior is undefined if the view doesn't have a predicate.



Remarks

For 1, the Simple_view requirement means that a view `V` and `const V` have the same iterator and sentinel types.

end

Get the sentinel at the end of the view.

```
C++
```

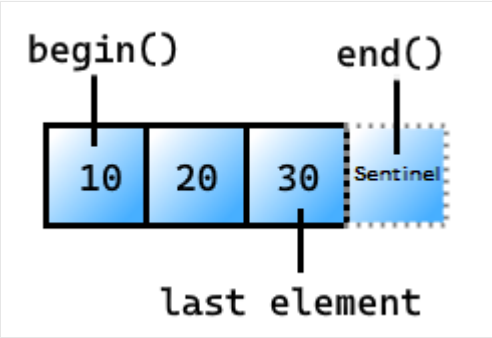
```
1) constexpr auto end() requires (!Simple_view<V>);
2) constexpr auto end() const requires
        range<const V> &&
        indirect_unary_predicate<const Pred, iterator_t<const V>
```

Parameters

None.

Return value

The sentinel that follows the last element in the view.



Remarks

For 1, the Simple_view requirement means that a view `V` and `const V` have the same iterator and sentinel types.

`pred`

Get a reference to the predicate used to select which leading elements will go in the view.

C++

```
constexpr const Pred& pred() const;
```

Return value

A reference to the predicate used to select the leading elements to put in the view.

Example `pred`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
```

```
#include <vector>

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto mv = v | std::views::take_while(
        [](int i) {return i < 5; });
    std::cout << std::boolalpha << mv.pred()(v[6]); // outputs false because
v[6] = 6 and 6 is not less than 5 (the predicate)
}
```

See also

<ranges>

take_view

take_while range adaptor

view classes

`transform_view` class (C++ Standard Library)

Article • 02/07/2023

A view of elements, each of which is a transformation of an element in the specified range.

Syntax

```
C++
```

```
template<input_range V, move_constructible F>
    requires view<V> && is_object_v<F> &&
    regular_invocable<F&, range_reference_t<V>> &&
    can-reference<invoke_result_t<F&, range_reference_t<V>>>
class transform_view : public view_interface<transform_view<V, F>>;
```

Template parameters

F

The type of the function object that transforms the elements.

V

The type of the underlying view.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::transform
Underlying range	Must satisfy input_range or higher
Element type	Same as the transformation function's return type.
View iterator category	Supports input_range up to random_access_range, depending on the underlying range
Sized	Only if the underlying range satisfies sized_range

Characteristic	Description
Is `const`-iterable	Only if the underlying range is `const` iterable and the transformation works on `const` references.
Common range	Only if the underlying range satisfies common_range
Borrowed range	No

Members

Member functions	Description
Constructors[C++20]	Construct the view.
base[C++20]	Get the underlying range.
begin[C++20]	Get an iterator to the first element.
end[C++20]	Get the sentinel at the end of the view.
size[C++20]	Get the number of elements. The underlying range must satisfy sized_range.
Inherited from view_interface	**Description**
back[C++20]	Get the last element.
empty[C++20]	Test whether the view is empty.
front[C++20]	Get the first element.
operator bool[C++20]	Test whether the view isn't empty.
operator[][C++20]	Get the element at the specified position.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Constructors

Construct an instance of a `transform_view`

```
C++

1) transform_view() requires default_initializable<V>
         && default_initializable<F> = default;
2) constexpr transform_view(V base, F func);
```

Parameters

`base`
The underlying view.

`func`
The function that transforms each element.

For information about template parameter types, see Template parameters.

Return value

A `transform_view` instance.

Remarks

The best way to create a `transform_view` is by using the views::transform range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1) Create a value-initialized `transform_view`. The transformation function and the underlying view must be default-initializable.
2) Move construct the `transform_view` from a *`base`* view and a transformation function *`func`*. Both *`base`* and *`func`* are moved via `std::move()`.

Example: `transform_view`

```
C++

// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>
#include <chrono>
```

```
using namespace std;
using namespace chrono;

void print(auto v)
{
    for (auto x : v)
    {
        cout << x << ' ';
    }
    cout << '\n';
}

struct classes
{
    string className;
    weekday startDay;
};

int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};

    // outputs 0 2 4 6 -8 10 12
    print(v | std::views::transform([](int i) {return i * 2; }));

    // ---- Modify the elements in the collection by returning a reference
to the element to transform

    std::vector<classes> theClasses = {
        {"Math", Monday},
        {"English", Wednesday},
        {"History", Monday},
        {"Science", Wednesday},
        {"Art", Friday},
        {"Music", Thursday}
    };

    // lambda to get a reference to the the day of the week for a class
    auto getDay = [](classes& c) -> weekday&
    {
        return c.startDay;
    };

    // If a class starts on Monday, change it to Tuesday
    for (auto&& startDay : theClasses | std::views::transform(getDay))
    {
        // modify the startDay in the collection
        if (startDay == Monday)
        {
            startDay = Tuesday;
        }
    }

    // output classes and start times
    for (auto c : theClasses)
```

```
    {
        std::cout << c.className << " : " << c.startDay << '\n';
    }
}
```

Output

```
0 2 4 6 -8 10 12
Math : Tue
English : Wed
History : Tue
Science : Wed
Art : Fri
Music : Thu
```

base

Get the underlying view.

C++

```
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses std::move() to return the underlying view
constexpr V base() &&;
```

Parameters

None.

Returns

The underlying view.

begin

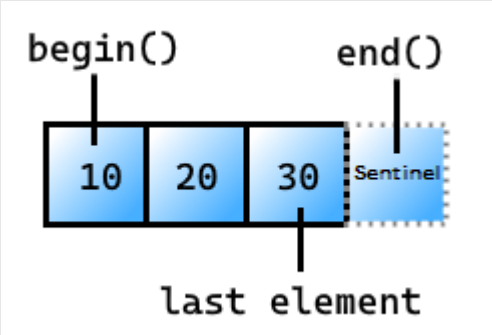
Get an iterator to the first element in the view.

C++

```
constexpr auto begin();
```

Return value

An iterator pointing at the first element in the view. The behavior is undefined if the view doesn't have a predicate.



`end`

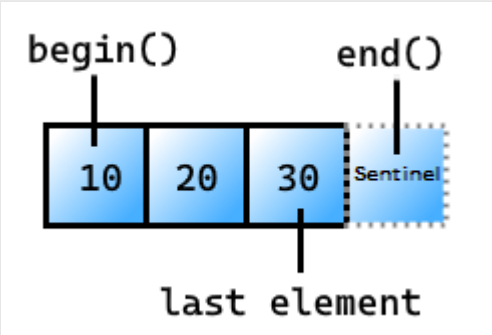
Get the sentinel at the end of the view.

```
C++
```

```
constexpr auto end()
```

Return value

The sentinel that follows the last element in the view:



`size`

Get the number of elements in the view.

```
C++
```

```
constexpr auto size() requires ranges::sized_range<V>;
constexpr auto size() const requires ranges::sized_range<const V>;
```

Parameters

None.

Return value

The number of elements in the view.

See also

<ranges>
filter range adaptor
view classes

`values_view` class (C++ Standard Library)

Article • 02/07/2023

A view consisting of the second index into each tuple-like value in a collection. For example, given a range of `std::tuple<string, int>` values, create a view consisting of all the `int` elements from each tuple.

`values_view` is an alias for `elements_view<R, 1>` and is useful for creating a view of the values in associative containers such as std::unordered_map.

Syntax

```
C++

template<input_range R>
using values_view = ranges::elements_view<R, 1>;
```

Template parameters

`R`

The type of the underlying range. This type must satisfy `ranges::input_range`.

View characteristics

For a description of the following entries, see View class characteristics

Characteristic	Description
Range adaptor	views::values
Underlying range	Must satisfy forward_range or higher
Element type	Same as the type of the second tuple element
View iterator category	`forward_range`, bidirectional_range, or random_access_range
Sized	Only if the underlying range satisfies sized_range
Is `const`-iterable	Only if the underlying range satisfies `const-iterable`

Characteristic	Description
Common range	Only if the underlying range satisfies common_range
Borrowed range	Only if the underlying range satisfies borrowed_range

Members

Member functions	Description
Constructors C++20	Construct a `values_view`.
base C++20	Get the underlying range.
begin C++20	Get an iterator to the first element.
end C++20	Get the sentinel at the end of the `values_view`.
size C++20	Get the number of elements.
Inherited from view_interface	**Description**
back C++20	Get the last element.
empty C++20	Test whether the `values_view` is empty.
front C++20	Get the first element.
operator[] C++20	Get the element at the specified position.
operator bool C++20	Test whether the `values_view` isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

Remarks

Tuple-like types that you can use with `values_view` are std::tuple, std::pair, and std::array.

Constructors

Construct an instance of a `values_view`.

```
C++

1) constexpr values_view(R base);
2) values_view() requires default_initializable<V> = default;
```

Parameters

`base`

The underlying range of tuple-like types.

For information about the template parameter type, see Template parameters.

Return value

An `values_view` instance.

Remarks

The best way to create an `values_view` is by using the values range adaptor. Range adaptors are the intended way to create view classes. The view types are exposed in case you want to create your own custom view type.

1. Create an `values_view` from the specified view.
2. The default constructor creates a default-constructed `values_view`.

Example: `values_view`

```
C++

// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <map>
#include <string>
#include <utililty>
#include <vector>

int main()
{
    // ========== work with a std::map

    std::map<std::string, int> cpp_standards
```

```
    {
        {"C++98", 1988},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"c++20", 2020}
    };

    // Extract all of the values from the map
    for (int years : std::views::values(cpp_standards))
    {
        std::cout << years << ' '; // 2003 2011 2014 2017 1988 2020
    }
    std::cout << '\n';

    // ========== work with a std::pair

    std::vector<std::pair<std::string, int>> windows
    {
        {"Windows 1.0", 1985},
        {"Windows 2.0", 1987},
        {"Windows 3.0", 1990},
        {"Windows 3.1", 1992},
        {"Windows NT 3.1", 1993},
        {"Windows 95", 1995},
        {"Windows NT 4.0", 1996},
        {"Windows 95", 1995},
        {"Windows 98", 1998},
        {"Windows 1.0", 1985},
        {"Windows 2000", 2000}
    };

    // Another way to call the range adaptor using '|': create a keys_view
from each pair
    for (int years : windows | std::views::values)
    {
        std::cout << years << ' '; // 1985 1987 1990 1992 ...
    }
}
```

Output

```
2003 2011 2014 2017 1988 2020
1985 1987 1990 1992 1993 1995 1996 1995 1998 1985 2000
```

base

Gets a copy of the underlying view.

```
```

```C++
// Uses a copy constructor to return the underlying view
constexpr V base() const& requires std::copy_constructible<V>;

// Uses a move constructor to return the underlying view
constexpr V base() &&;
```

Parameters

None

Return value

The underlying view.

`begin`

Get an iterator to the first element in the `values_view`.

```C++
// returns a non-const iterator
1) constexpr auto begin() requires (!Simple_view<V>);

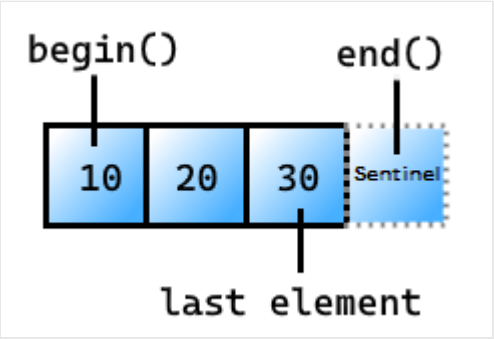
// returns a const iterator
2) constexpr auto begin() const requires ranges::range<const V>;
```

Parameters

None.

Return value

An iterator pointing at the first element in the `values_view`.



end

Get the sentinel at the end of the `values_view`

C++

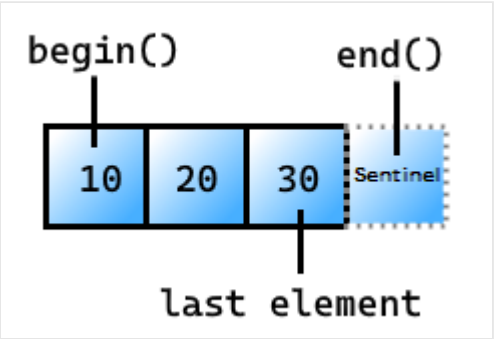
```
1) constexpr auto end() requires (!Simple_view<V> &&
!ranges::common_range<V>);
2) constexpr auto end() requires (!Simple_view<V> &&
ranges::common_range<V>);
3) constexpr auto end() const requires ranges::range<const V>;
4) constexpr auto end() const requires ranges::common_range<const V>;
```

Parameters

None.

Return value

The sentinel that follows the last element in the `values_view`



size

Get the number of elements.

C++

```
constexpr auto size() requires sized_range<V>
constexpr auto size() const requires sized_range<const V>
```

Parameters

None.

Return value

The number of elements in the `values_view`.

Remarks

The size of the view is only available if the underlying range is a sized_range, or in other words, bounded.

See also

elements_view
keys_view
<ranges>
View classes

`view_interface` class (C++ Standard Library)

Article • 02/07/2023

The base class for the view classes in the `std::ranges` namespace. This class implements some of the interface for derived view types. Use this as the base class for your own view types to reduce the boilerplate you need to write.

Syntax

```
C++
```

```
template<class Derived>
  requires std::is_class_v<Derived> &&
  std::same_as<Derived, std::remove_cv_t<Derived>>
class view_interface;
```

Template parameters

`Derived`

The type of the class that is deriving from this base class.

Members

Member functions	Description
back[C++20]	Get the last element in the derived view.
data[C++20]	Get a pointer to the first element in the derived view.
empty[C++20]	Test whether the derived view is empty.
front[C++20]	Get the first element in the derived view.
size[C++20]	Get the number of elements in the derived view.
Operators	**Description**
operator[][C++20]	Get the element at the specified position.
operator bool[C++20]	Test whether the derived view isn't empty.

Requirements

Header: `<ranges>` (since C++ 20)

Namespace: `std::ranges`

Compiler Option: /std:c++20 or later is required.

back

Get the last element in the derived view.

C++

```
constexpr auto back()
    requires ranges::bidirectional_range<Derived> &&
    ranges::common_range<Derived>;

constexpr auto back() const
    requires ranges::bidirectional_range<const Derived> &&
    ranges::common_range<const Derived>;
```

Parameters

None.

Return value

The last element in the derived view.

Remarks

The derived view must satisfy bidirectional_range and common_range.
The behavior of `back()` and `front()` are undefined for any empty view.

data

Get a pointer to the first element in the derived view.

C++

```
constexpr auto data()
    requires std::contiguous_iterator<ranges::iterator_t<Derived>>;
constexpr auto data() const
    requires ranges::range<const Derived> &&
    std::contiguous_iterator<ranges::iterator_t<const Derived>>;
```

Parameters

None.

Return value

A pointer to the first element in the derived view.

Remarks

The iterator for the derived view must satisfy contiguous_iterator.

`empty`

Test whether the derived view is empty.

C++

```
1) constexpr bool empty() requires ranges::forward_range<Derived>;
2) constexpr bool empty() const requires ranges::forward_range<const
Derived>;
```

Parameters

None.

Return value

Returns `true` if the derived view has no elements. Otherwise, returns `false`.

Remarks

The derived view must satisfy `std::ranges::forward_range`.

`front`

Get the first element in the derived view.

```
C++
```

```
constexpr auto front()
    requires ranges::forward_range<Derived>;
constexpr auto front() const
    requires ranges::forward_range<const Derived>;
```

Parameters

None.

Return value

The last element in the derived view.

Remarks

The derived view must satisfy forward_range.
The behavior of `front()` is undefined for `std::ranges::empty_view`.

`size`

Get the number of elements in the derived view.

```
C++
```

```
constexpr auto size() requires ranges::forward_range<Derived> &&
    std::sized_sentinel_for<ranges::sentinel_t<Derived>,
    ranges::iterator_t<Derived>>;
constexpr auto size() const requires ranges::forward_range<const Derived> &&
    std::sized_sentinel_for<ranges::sentinel_t<const Derived>,
    ranges::iterator_t<const Derived>>;
```

Parameters

None.

Return value

The number of elements in the derived view.

Remarks

The iterator for the derived view must satisfy sized_sentinel_for.

`operator[]`

Get the element at the specified position.

C++

```
template<ranges::random_access_range R = Derived>
constexpr decltype(auto) operator[](ranges::range_difference_t<R> pos);

template<ranges::random_access_range R = const Derived>
constexpr decltype(auto) operator[](ranges::range_difference_t<R> pos)
const;
```

Parameters

pos
The position, relative to the beginning iterator, of the element to return.

Return value

The element at the specified position relative to the beginning iterator.

Remarks

The derived view must satisfy random_access_range.
The behavior of this operator is undefined for `std::ranges::empty_view`.

Example: `operator[]`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
```

```cpp
#include <vector>

int main()
{
    std::vector<int> v{1, 2, 3, 4, 5};
    std::ranges::drop_view dv = std::views::drop(v, 2);

    for (auto e : dv)
    {
        std::cout << e << ' '; // 3 4 5
    }

    std::cout << "\ndv[1] = " << dv[1];
}
```

Output

```
3 4 5
dv[1] = 4
```

`view_interface::operator bool`

Test whether the derived view isn't empty.

C++

```cpp
explicit constexpr operator bool();
explicit constexpr operator bool() const;
```

Parameters

None.

Return value

Returns `false` if the derived view has no elements (the view is empty). Otherwise, returns `true` (the view isn't empty).

Remarks

The iterator for the derived view must satisfy `std::ranges::forward_iterator`.
This operator is equivalent to `!empty()`. This makes it convenient to write `if`

`(someRange) {...}` to test whether there's something in the range to operate on. The behavior of this operator is undefined for `std::ranges::empty_view`.

Example: `operator bool`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{1, 2, 3, 4, 5};
    std::ranges::filter_view fv = std::views::filter(v, [](int e) { return e
> 3; });

    bool isNotEmpty = static_cast<bool>(fv);
    std::cout << "Has elements greater than 3: " << std::boolalpha <<
isNotEmpty << '\n' >>;
}
```

Output

```
Has elements greater than 3: true
```

See also

<ranges>
ranges::begin()
ranges::data()
ranges::end()
ranges::empty()
ranges::size()
View classes

`<ranges>` alias templates

Article • 01/30/2023

An alias template is an alias for another type, which can make code more readable. For example, the following alias, `conditional_t`, is an alias for either `borrowed_range` or `dangling` range, depending on the kind of `range` that's passed in:

C++

```
// requires /std:c++20, or later

#include <iostream>
#include <list>
#include <span>
#include <algorithm>
#include <ranges>
#include <type_traits>

using namespace std;

// Define an alias template called my_iterator_t
// If the provided range R is a borrowed_range, then the
// returned type is iterator_t<R>; otherwise, ranges::dangling
template<ranges::range R>
using my_iterator_t = conditional_t<
    ranges::borrowed_range<R>,
    ranges::iterator_t<R>, ranges::dangling>;

int main()
{
    my_iterator_t<list<int>> aDanglingRange; // list<> isn't a
borrowed_range
    constexpr bool same = same_as<
        decltype(aDanglingRange),
        ranges::dangling>; // true

    my_iterator_t<span<int, 5>> anIterator_t; // span<> is a borrowed_range
    constexpr bool same2 = same_as<
        decltype(anIterator_t),
        ranges::iterator_t<span<int, 5>>>; // true

    cout << boolalpha << same << "," << same2; // outputs true, true
}
```

For more information about alias templates, see Aliases and typedefs.

The `<algorithm>` header defines the following alias templates that determine the types of iterators and sentinels for a `range`:

Alias template	Description
borrowed_iterator_t[C++20]	Determine if an iterator returned for a `range` refers to a range whose lifetime has ended.
borrowed_subrange_t[C++20]	Determine if a `subrange` returned for a `range` refers to a range whose lifetime has ended.
dangling[C++20]	Indicates that the returned iterator of a `range`/`subrange` outlives the lifetime of the `range`/`subrange` it refers to.
iterator_t[C++20]	Returns the iterator type for the specified range.
range_difference_t[C++20]	Returns the difference type for the specified range's iterator.
range_reference_t[C++20]	Returns the reference type for the specified range's iterator.
range_rvalue_reference_t[C++20]	Returns the rvalue reference type for the specified range's iterator. In other words, the rvalue reference type of the range's elements.
range_size_t[C++20]	Returns the type used to report the specified range's `size`.
range_value_t[C++20]	Returns the value type of specified range's iterator. Or in other words, the type of the elements in the range.
sentinel_t[C++20]	Returns the sentinel type for the specified range.

`borrowed_iterator_t`

When an algorithm function that returns an iterator is called with an rvalue `range` argument, the range's lifetime could end following the call. That means the returned iterator could refer to elements whose lifetimes have ended. Using a dangling iterator results in undefined behavior.

This template alias returns `ranges::dangling` to indicate that this is the situation for the given range argument, or `std::ranges::iterator_t<R>` to indicate that it's safe to use the returned iterator because the range it refers to models `borrowed_range` or the range was passed as an lvalue.

C++

```
template<ranges::range R>
using borrowed_iterator_t = conditional_t<ranges::borrowed_range<R>,
    ranges::iterator_t<R>, ranges::dangling>;
```

Parameters

R

The range to test.

Remarks

The lifetime of an rvalue range can end following a function call whether the range models `borrowed_range` or not. If it's a `borrowed_range`, you may be able to continue to use the iterators with well-defined behavior regardless of when the range's lifetime ends.

Cases where this isn't true are, for example, for containers like `vector` or `list` because when the container's lifetime ends, the iterators would refer to elements that have been destroyed.

You can continue to use the iterators for a `borrowed_range`, for example, for a `view` like `iota_view<int>{0, 42}` whose iterators are over set of values that aren't subject to being destroyed because they're generated on demand.

If an algorithm function is passed a range whose iterators depend on its lifetime, `ranges::dangling` is returned instead of an iterator or subrange, so potential misuse is detected at compile time.

Example: `borrowed_iterator_t`

The following example shows how `borrowed_iterator_t` detects a dangling iterator. The function `ranges::max_element()` uses this template alias to determine the return type:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <span>
#include <ranges>
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    // Not dangling ------------------
```

```
    int a[] = {0,1,2,3};

    // not dangling even though an rvalue because span models
ranges::borrowed
    auto result1 = ranges::max_element(span{a});
    cout << boolalpha << ranges::borrowed_range<decltype(span{a})> << endl;
// outputs true because the temporary models ranges::borrowed
    cout << same_as<decltype(result1), ranges::dangling> << endl; // outputs
false because the result isn't dangling

    vector<int> v{0,1,2,3}; // doesn't model ranges::borrowed
    auto result2 = ranges::max_element(v); // Yet not dangling because
passed as an lvalue
    cout << same_as<decltype(result2), ranges::dangling> << endl; // outputs
false because the result isn't dangling

    // Dangling -----------------

    auto result3 = ranges::max_element(vector{0,1,2,3}); // dangling because
vector doesn't model ranges::borrowed and is passed as an rvalue
    cout << same_as<decltype(result3), ranges::dangling>; // outputs true
because the result is dangling
}
```

Output

```
true
false
false
true
```

`borrowed_subrange_t`

When an algorithm function that returns a `subrange` is called with an rvalue `range` argument, the range's lifetime could end following the call. That means the returned `subrange` could refer to elements whose lifetimes have ended. Using a dangling `subrange` results in undefined behavior.

This template alias either returns `ranges::dangling` to indicate that this could be the situation for the given range argument, or `subrange<ranges::iterator_t<R>>` to indicate that it's safe to use the returned subrange because either the range whose elements it refers to models `borrowed_range` or the range was passed as an lvalue.

C++

```
template<ranges::range R>
using borrowed_subrange_t = conditional_t<ranges::borrowed_range<R>,
```

```
    ranges::subrange<ranges::iterator_t<R>>, ranges::dangling>;
```

Parameters

R

The range to test.

Remarks

The lifetime of an rvalue range can end following a function call whether the range models `borrowed_range` or not. If it's a `borrowed_range`, you may be able to continue to use the iterators with well-defined behavior regardless of when the range's lifetime ends.

Cases where this isn't true are, for example, for containers like `vector` or `list` because when the container's lifetime ends, the iterators would refer to elements that have been destroyed.

You can continue to use the iterators for a `borrowed_range`, for example, for a `view` like `iota_view<int>{0, 42}` whose iterators are over set of values that aren't subject to being destroyed because they're generated on demand.

If an algorithm function is passed a range whose iterators depend on its lifetime, `ranges::dangling` is returned instead of a subrange so that potential misuse is detected at compile time.

Example: `borrowed_subrange_t`

The following example shows how `borrowed_subrange_t` detects a dangling iterator because `equal_range()` and `max_element` use this template alias to determine the return type:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <iostream>
#include <algorithm>
#include <span>
#include <ranges>

int main()
```

```
{
    using namespace std;

    // Not dangling ------------------

    vector vec{0, 1, 1, 2};

    auto result1 = ranges::equal_range(span{vec}, 1); // not dangling even
though passing as an rvalue because span models borrowed_range
    cout << boolalpha << ranges::borrowed_range<decltype(span{vec})> <<
endl;  // true because the temporary is a borrowed range
    cout << boolalpha << same_as<decltype(result1), ranges::dangling> <<
endl; // false because the result isn't dangling

    // result2 isn't dangling even though vec doesn't model ranges::borrowed
because it's an lvalue
    auto result2 = ranges::max_element(vec);
    cout << boolalpha << ranges::borrowed_range<decltype(vec)> << endl;  //
false because vector isn't a borrowed_range
    cout << boolalpha << same_as<decltype(result2), ranges::dangling> <<
endl; // false because the result isn't dangling

    // Dangling ----------------------

    // result3 is dangling because the temporary is an rvalue that doesn't
model borrowed_range
    auto result3 = ranges::max_element(vector{0,1,1,2});
    cout << boolalpha << same_as<decltype(result3), ranges::dangling> <<
endl; // true because the result is dangling
}
```

Output

```
true
false
false
false
true
```

`dangling`

If an algorithm function that returns an iterator or a `subrange` is called with an rvalue `range` argument, the range argument's lifetime could end following the call. That means the returned iterator or `subrange` could refer to elements whose lifetimes have ended. Using a dangling iterator or `subrange` results in undefined behavior.

If an algorithm function is passed a range whose iterators depend on its lifetime, `ranges::dangling` is returned instead of an iterator or subrange so that potential misuse

is detected at compile time.

C++

```
1) constexpr dangling() noexcept = default;
2) template<class... Args>
constexpr dangling(Args&&...) noexcept {}
```

Parameters

Args

A variable number of non-`void` types. They have no effect. The arguments are a convenience so that you don't need different code paths to handle constructing the iterator type versus the `dangling` type. This is useful when the passed in value indicates that `dangling` should be returned instead of an iterator.

Example: `dangling`

The following example shows how `max_element` detects a dangling iterator.

C++

```
// requires /std:c++20, or later

#include <vector>
#include <iostream>
#include <ranges>
#include <algorithm>

using namespace std;

int main()
{
    auto result1 = ranges::max_element(vector{1,2,3}); // dangling because
vector doesn't model ranges::borrowed and is passed as an rvalue
    cout << boolalpha << same_as<decltype(result1), ranges::dangling> <<
endl; // outputs true because the result is dangling

    vector<int> v{3,4,5};
    auto result2 = ranges::max_element(v); // Not dangling because passed as
an lvalue
    cout << same_as<decltype(result2), ranges::dangling>; // outputs false
because the result isn't dangling
}
```

Output

```
true
false
```

`iterator_t`

This template alias returns the iterator type used to iterate over the provided range type.

C++

```
template<class T>
using iterator_t = decltype(ranges::begin(declval<T&>()));
```

Parameters

T

The range type to get the iterator type for.

Example: `iterator_t`

The following example shows how `iterator_t` can be used to declare an iterator for a vector:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};

    ranges::iterator_t<decltype(v)> it = v.begin();
    cout << *it << "\n"; // outputs 1
    cout << typeid(it).name(); // outputs class _Vector_iterator<class
_Vector_val<struct _Simple_types<int>>>
}
```

Output

```
1
class std::_Vector_iterator<class std::_Vector_val<struct
std::_Simple_types<int> > >
```

`range_difference_t`

Returns the difference type for the specified range's iterator.

C++

```
template<range R>
using range_difference_t = iter_difference_t<iterator_t<R>>;
```

Parameters

R

The range whose iterator will provide the difference type.

Example: `range_difference_t`

The following example shows how `range_difference_t` is used to hold the distance between elements in a range:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};

    auto findIt = ranges::find(v, 2);
    // type of distance is ptrdiff_t
    ranges::range_difference_t<decltype(v)> distance =
ranges::distance(v.begin(), findIt);
    cout << distance << endl; // outputs 1
}
```

Output

```
1
```

`range_reference_t`

Returns the reference type for the specified range's iterator. In other words, the reference type of the range's elements.

C++

```
template <range R>
using range_reference_t = iter_reference_t<ranges::iterator_t<R>>;
```

Parameters

R

The range for which the reference type of its iterator type is returned.

Example: `range_reference_t`

The following example shows `range_reference_t` referring to the type of the elements in a range:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};

    ranges::range_reference_t<decltype(v)> ref = v[0];

    cout << ref << endl; // outputs 1
    cout << typeid(ref).name() << endl; // outputs int
}
```

Output

```
1
int
```

`range_rvalue_reference_t`

Returns the rvalue reference type for the specified range's iterator. In other words, the rvalue reference type of the range's elements.

C++

```
template <range R>
using range_rvalue_reference_t = iter_reference_t<ranges::iterator_t<R>>;
```

Parameters

`R`

The range to get the rvalue reference type to its iterator type.

Example: `range_rvalue_reference_t`

The following example shows `range_rvalue_reference_t` referring to a rvalue type of the elements in a range:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};

    ranges::range_rvalue_reference_t<decltype(v)> elementRvalueType = v[0] *
10; // elementRvalueType is int&&

    cout << elementRvalueType << endl; // outputs 10
    cout << typeid(elementRvalueType).name() << endl; // outputs int
}
```

Output

```
10
int
```

`range_size_t`

Returns the type of the `size` function for the specified sized_range.

C++

```
template <range R>
using range_size_t = iter_reference_t<ranges::iterator_t<R>>;
```

Parameters

`R`

The range to get the type of its `size` function.

Example: `range_size_t`

The following example shows `range_size_t` referring to the number of elements in a range:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <iostream>
#include <ranges>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};

    ranges::range_size_t<decltype(v)> size = v.size();
    cout << size << endl; // outputs 3
    cout << typeid(size).name(); // outputs unsigned __int64
}
```

Output

```
3
unsigned __int64
```

`range_value_t`

Returns the value type of specified range's iterator. Or in other words, the type of the elements in the range.

C++

```
template <ranges::range R>
using range_value_t = iter_value_t<ranges::iterator_t<R>>;
```

Parameters

R

The range to get the value type of its iterator.

Example: `range_value_t`

The following example shows how `range_value_t` refers to the type of elements in a range:

C++

```
// requires /std:c++20, or later

#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    using namespace std;

    vector<int> v{1,2,3};
    ranges::range_value_t<decltype(v)> elementType = v[2]; // elementType is
an int

    cout << elementType << endl; // outputs 3
    cout << typeid(elementType).name() << endl; // outputs int
}
```

Output

```
3
unsigned int
```

`sentinel_t`

Returns the sentinel type for the specified range.

C++

```
template <range R>
using sentinel_t = decltype(ranges::end(declval<R&>()));
```

Parameters

R

The range to get the sentinel type for.

Example: `sentinel_t`

The following example shows using `sentinel_t` to determine whether the iterator type and sentinel type are the same:

C++

```
// requires /std:c++20, or later

#include <iostream>
#include <list>
#include <ranges>

int main()
{
    using namespace std;

    list myList{1, 2, 3};
    ranges::subrange count = std::views::counted(myList.begin(),
myList.size());

    ranges::iterator_t<decltype(count)> first;
    ranges::sentinel_t<decltype(count)> last;

    // The iterator type and the sentinel type of a subrange
    // obtained from views::counted are not the same
    cout << boolalpha << is_same<decltype(first), decltype(last)>::value <<
endl; // outputs false
    cout << "iter: " << typeid(first).name() << "\n\n end: " <<
```

```
typeid(last).name() << endl;
}
```

Output

```
false
iter: class std::counted_iterator<class std::_List_iterator<class
std::_List_val<struct std::_List_simple_types<int> > > >

 end: struct std::default_sentinel_t
```

See also

<ranges>

Range adaptors

View classes

Range adaptors

Article • 12/21/2022

Range adaptors create a *view* (one of the View classes in the `std::views` namespace) from a range. We recommend that you use an adaptor to create views instead of creating the view types directly. The adaptors are the intended way to access views. They're easier to use, and in some cases more efficient, than creating instances of the view types directly.

A view is a lightweight object that refers to elements from a range. A view can:

- Consist of only certain elements from a range.
- Represent a transformation of elements from a range.
- Be the reverse of, or only the first `n` elements of, a range.
- Be a combination of the preceding things.

A view is cheap, `O(1)`, to copy, assign, and destroy--no matter how many elements are involved. Consider the following example:

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> input =  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto divisible_by_three = [](const int n) {return n % 3 == 0;};
    auto square = [](const int n) {return n * n;};

    auto x = input | std::views::filter(divisible_by_three)
                   | std::views::transform(square);

    for (int i : x)
    {
        std::cout << i << ' ';
    }
}
```

Output

```
0 9 36 81
```

The first range adaptor, filter, provides a view that contains the elements from `input` that are divisible by three. The other range adaptor, transform, takes the view that contains the elements divisible by three and provides a view of the square of those elements.

When a range adaptor produces a view, it doesn't incur the cost of transforming every element in the range to produce that view. The cost to process an element in the view is paid only when you access that element.

Creating a view is preparation to do work in the future. In the previous example, creating the view doesn't result in finding all the elements divisible by three or squaring those elements. Work happens only when you access an element in the view.

Elements of a view are usually the actual elements of the range used to create the view. The view usually doesn't own the elements; it just refers to them, with the exception of owning_view. Changing an element changes that element in the range that the view was created from. The following example shows this behavior:

C++

```
#include <algorithm>
#include <iostream>
#include <ranges>

int main()
{
    int input[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto even = [](const int n) { return n % 2 == 0; };
    auto x = input | std::views::filter(even); // create a view of the even
elements from input

    for (int &i : x)
    {
        std::cout << i << ' '; // 0 2 4 6 8 10
    }
    std::cout << '\n';

    std::ranges::fill(x, 42); // changes the evens from input[] to 42
    for (int &i : input) // demonstrates that the even elements in the range
are modified
    {
        std::cout << i << ' '; // // 42 1 42 3 42 5 42 7 42 9 42
    }
}
```

Range adaptors come in many forms. For example, there are range adaptors that allow you to produce a view by:

- Filtering another range based on a predicate (filter).
- Transforming the elements in a range (transform).
- Splitting a range (split).

Range adaptors can be chained together (composed). That's where the power and flexibility of ranges are most apparent. Composing range adaptors allows you to overcome a core problem with the previous Standard Template Library (STL) algorithms, which is that they aren't easy to chain together.

The following range adaptors are available in the `std::views` namespace. The `std::views` namespace is a convenience alias for `std::ranges::views`.

Range adaptor	Description
all C++20	Create a view that refers to a range and its elements.
common C++20	Create a view that has the same iterator and sentinel types from a range that doesn't.
counted C++20	Create a view of the first *n* elements of a range, starting from the specified location.
drop C++20	Create a view from another view, skipping the specified number of elements from the front.
drop_while C++20	Create a view that contains the elements of a range that remain after the leading elements that match the specified condition are dropped.
elements C++20	Create a view of the selected index into each tuple-like value in a range.
empty C++20	Create a view that has no elements.
filter C++20	Create a view that contains the elements of a range that match the specified condition.
iota C++20	Create a view that contains a sequence of increasing values.
istream C++20	Create a view over the elements of a stream.
join C++20	Create a view that combines all the elements of multiple ranges into a single view.
keys C++20	Create a view of the first index into each tuple-like value in a collection.
lazy_split C++20	Split a view into subranges based on a delimiter.
reverse C++20	Create a view of the elements of a range in reverse order.
single C++20	Create a view that contains one element.

Range adaptor	Description
split[C++20]	Split a view into subranges based on a delimiter.
take[C++20]	Create a view of the first *n* elements from another view.
take_while[C++20]	Create a view that contains the leading elements of a range that match the specified condition.
transform[C++20]	Create a view of transformed elements from another view.
values[C++20]	Create a view of the second index into each tuple-like value in a collection.

In the previous table, a range adaptor is typically described as taking a range and producing a view. To be precise, range adaptors have a range argument that accepts one of the following:

- The `cv-unqualified` type models view, and the argument is an rvalue or is copyable.
- When you pass the argument as an lvalue, it must model range and live as long as the view.
- When you pass the argument as an rvalue, such as when calling owning_view, it must model `range` and `movable`.

Range adaptor functions are typically function objects, which look like function calls and enforce constraints on the types that can be passed.

You can pass range adaptors and the result of pipe operations (`|`) to code that expects function objects. In the following example, the view that the `split` range adaptor creates is passed to the `transform` range adaptor as if by a function call, because the `transform` range adaptor is a function object.

C++

```
std::map<int, string> x = {{0, "Hello, world"}, {42, "Goodbye, world"}};
auto y = x | views::values | views::transform(views::split(' '));
// y is a range whose elements are ranges of whitespace-delimited strings
from each value in x:
// {{"Hello", "world"}, {"Goodbye", "world"}}
```

`all`

Create a view of all the elements in a range.

C++

```
template <ranges::viewable_range R>
constexpr ranges::view auto all(R&& rg) const noexcept;
```

Parameters

`R`
The type of the underlying range.

`rg`
The range to create the view from.

Return value

- If `rg` is already a view, a copy of `rg`.
- If `rg` is a non-view lvalue, a ref_view that refers to `rg`. (The lifetime of the view is tied to the lifetime of `rg`.)
- If `rg` is a non-view rvalue such as a temporary object, or is the result of passing the range to `std::move`, an owning_view.

Use `std::views::all_t<decltype((rg))>` to get the type of the returned view.

Remarks

This range adaptor is the best way to convert a range into a view. One reason to create a view from a range is to pass it by value at low cost, if passing the range by value could be expensive.

Getting a view for a range is a useful alternative to passing a heavyweight range by value because views are inexpensive to create, copy, and destroy. A possible exception is `owning_view`, which is a view that owns the underlying range.

In general, the worst-case scenario for destroying a view has `O(N)` complexity for the number of elements in the range. Even if you destroy `K` copies of view with `N` elements, the total complexity is still `O(N)` because the underlying range is destroyed only once.

Example: `all`

C++

```
// requires /std:c++20 or later
#include <ranges>
```

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};
    auto myRefView = std::views::all(v); // create a ref_view of the vector
    std::cout << myRefView.size() << '\n'; // 10

    auto myOwningView = std::views::all(std::move(v)); // create an
owning_view from a moved vector
    std::cout << myRefView.size() << '\n'; // outputs 0 because myOwningView
now owns the elements
    std::cout << v.size() << '\n'; // outputs 0 because myOwningView now
owns the elements
    std::cout << myOwningView.size(); // 10
}
```

Output

```
10
0
0
10
```

common

Create a view that has the same begin iterator and sentinel type from a range that might not.

C++

```cpp
template <ranges::viewable_range R>
constexpr ranges::view auto common(R&& rg) const noexcept;
```

Parameters

`R`

The type of the underlying range.

`rg`

The range to create the view from.

Return value

- `views::all(rg)` if `rg` is a range with the same iterator and sentinel type.
- common_view(views::all(rg)) if `rg` has different iterator and sentinel types.

Remarks

When an API requires the begin iterator and end sentinel to have the same type, and the view that you're using doesn't meet that requirement (or you don't know if it does), use this range adaptor to create a `common_view`. It guarantees that the type of the begin iterator and the type of the end sentinel are the same.

Example: `common`

C++

```
// requires /std:c++20 or higher
#include <ranges>
#include <iostream>
#include <numeric>
#include <list>

int main()
{
    std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

    auto firstFive = std::views::take(lst, 5);
    // firstFive.begin(), firstFive.end() have different types:
counted_iterator versus default_sentinel
    // auto r = std::accumulate(firstFive.begin(), firstFive.end(), 0); //
Error: accumulate() requires firstFive.begin() and firstFive.end() types to
be the same.

    auto common = std::views::common(firstFive); // create a common_view
that has the same begin/end iterator types
    std::cout << std::accumulate(common.begin(), common.end(), 0); // Now
you can call the API because the iterator types are the same. Outputs 15
(1+2+3+4+5)
}
```

Output

```
15
```

`counted`

Create a view of the first `count` elements of a range, starting from the specified location.

C++

```
template<class Iterator>
constexpr auto counted(Iterator&& it, iter_difference_t<Iterator> count);
```

Parameters

`DifferenceType`
The type of the count.

`Iterator`
The type of the iterator.

`count`
The number of elements to include in the view. Must be non-negative.

- If `count == 0`, an empty span is returned.
- If `count` is greater than the number of elements in the range, the behavior is undefined.

`it`
An iterator to the element in the range to start with. The element that the iterator points to is included in the created view.

Return value

A span is returned if `it` is a contiguous_iterator for arrays, vectors, and other containers that store their elements contiguously. Otherwise, a subrange is returned.

Remarks

The included elements are `[it, count)`.

After the view is created, the number of elements in the view stays the same, even if the range that it was created from changes. However, if the underlying range changes, accessing elements from the view might result in undefined behavior.

Example: `counted`

C++

```
// requires /std:c++20 or later
#include <algorithm>
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto pos5 = std::ranges::find(v, 5);
    auto countedView = std::views::counted(pos5, 5);
    for (auto e : countedView) // outputs 5 6 7 8 9
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';

    // You can pass the range directly if it supports
input_or_output_iterator, in which case
    // the count starts from the first element
    const char chars[] = { 'H','i',' ','t','h','e','r','e' };
    for (char c : std::views::counted(chars, 2))
    {
        std::cout << c; // outputs Hi
    }
}
```

Output

```
5 6 7 8 9
Hi
```

drop

Create a view that excludes the first *n* elements of a range.

C++

```
1) template<ranges::viewable_range R>
constexpr ranges::view auto drop(R&& rg, ranges::range_difference_t<R>
count);

2) template<class DifferenceType>
constexpr /* range closure object */ drop(DifferenceType&& count);
```

Parameters

`DifferenceType`

The type that describes the number of elements to skip.

`count`

The number of elements to drop from the front of `rg`. Must be non-negative.

- If `count == 0`, all the elements in `rg` are returned.
- If `count` is greater than the number of elements in `rg`, an empty view is returned.

`R`

The type of the range.

`rg`

The range that's used to create the view.

Return value

A view of the underlying range, with the specified number of elements dropped from the front.

If you specify more elements to drop than exist in the underlying range, an empty_view is returned.

The returned view is typically, but not always, a specialization of drop_view. That is:

- If `V` is a specialization of empty_view, or is a specialization of span, basic_string_view, iota_view, or subrange that is both random_access_range and sized_range, the result is a specialization of `V`.
- Otherwise, the result is a drop_view.

Remarks

After it's created, the number of elements in the view stays the same even if the view that it was created from changes. However, if the underlying view changes, accessing elements in the returned view might result in undefined behavior.

`drop` is the opposite of take.

The code shown earlier as "2)" can be used with pipe syntax: `collection | drop(5)`. Or it can be used with function call syntax: `drop(collection, 5)` or `drop(5)(collection)`.

Example: `drop`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{1, 2, 3, 4, 5};
    auto newView = std::views::drop(v, 3);
    for (auto e : newView) // 4 5
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';

    auto numbers = std::views::iota(0) | std::views::take(10); // build a
view of 10 integers
    auto latterHalf = numbers | std::views::drop(5);
    for (auto i : latterHalf)
    {
        std::cout << i << ' '; // 5 6 7 8 9
    }
}
```

Output

```
4 5
5 6 7 8 9
```

drop_while

Create a view that contains the elements of a range that remain after the leading elements that match the specified condition are dropped.

C++

```
1) template<ranges::viewable_range R, class P>
constexpr ranges::view auto drop_while(R&& rg, P&& predicate);

2) template<class P>
constexpr /*range adaptor closure*/ drop_while(P&& predicate);
```

Parameters

`R`

The type of the range.

`predicate`

The conditions that determine which leading elements to drop from the range.

`rg`

The underlying range to create the view from.

Return value

A drop_while_view that consists of the elements that remain when the leading elements that match the predicate are dropped.

Remarks

Stops dropping elements from `rg` as soon as the predicate returns `false`.

`drop_while` is the opposite of take_while.

The code shown earlier as "2)" can be used with pipe syntax: `collection | drop_while(predicate)`. Or it can be used with function call syntax: `drop_while(collection, predicate)` or `drop_while(predicate)(collection)`.

Example: `drop_while`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

void print(auto&& v)
{
    for (auto&& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto myView = std::views::drop_while(
```

```
        v,
        [](int i) {return i >= 0; });
    print(myView); // -4 5 6

    auto myView2 = v | std::views::drop_while(
        [](int i) {return i < 5; });
    print(myView2); // 5 6
}
```

Output

```
-4 5 6
5 6
```

`elements`

Create an elements_view, which is a view of the selected index into each tuple-like value in a range. For example, given a range of `std::tuple<string, int>` values, create an `elements_view` of all the `string` elements from each tuple.

C++

```
template<ranges::viewable_range R, size_t N>
constexpr ranges::view auto elements<N>(R&& rg);
```

Parameters

`N`
The index of the element to select from each tuple-like value to include in the view.

`R`
The type of the underlying range.

`rg`
The range of tuple-like values to create the view from.

Return value

An elements_view that consists of the selected index into each tuple-like value in a collection.

Example: `elements`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <map>
#include <string>

int main()
{
    std::map<std::string, int> cpp_standards
    {
        {"C++98", 1998},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"C++20", 2020}
    };

    // Create an elements_view of all the string elements from each tuple
    for (int const year : std::views::elements<1>(cpp_standards))
    {
        std::cout << year << ' '; // 2003 2011 2014 2017 1998 2020
    }
    std::cout << '\n';

    // Another way, using |: create an elements_view of all the int elements
from each tuple
    for (auto&& name : cpp_standards | std::views::elements<0>)
    {
        std::cout << name << ' '; // C++03 C++11 C++14 C++17 C++98 c++20
    }
}
```

Output

```
2003 2011 2014 2017 1998 2020
C++03 C++11 C++14 C++17 C++98 c++20
```

`empty`

Create an `empty_view`, which is a view that has no elements.

C++

```
template<class T>
inline constexpr empty_view<T> empty{};
```

Parameters

`T`

The type of the elements in the view. The view needs an element type, even though there are no elements.

Return value

An empty_view.

Remarks

An `empty_view` can be useful when you're calling code that requires a view but doesn't need to process any of its elements.

Example: `empty`

C++

```
// requires /std:c++20 or higher
#include <ranges>
#include <iostream>

int main()
{
    auto anEmptyView = std::views::empty<int>;
    bool isNotEmpty = (bool)anEmptyView;
    std::cout << boolalpha << isNotEmpty << "\n"; // false
}
```

Output

```
false
```

`filter`

Create a view that contains the elements of a range that match the specified condition.

C++

```
1) template<ranges::viewable_range R, class P>
    requires requires {filter_view(forward<R>(rg), forward<P>(predicate));}
constexpr ranges::view auto filter(R&& rg, P&& predicate);
```

```
2) template<class P>
constexpr /*range adaptor closure*/ filter(P&& predicate);
```

Parameters

`P`

The type of the predicate.

`predicate`

The conditions that determine which elements to keep in the range.

`R`

The type of the underlying range.

`rg`

The range to create the view from.

Return value

A filter_view that contains the elements of a range that match the predicate.

Remarks

For efficiency's sake, when you use `filter` and `transform` together with a pipe `|`, do the `filter` first so that you `transform` only the elements that you intend to keep.

The code shown earlier as "2)" can be used with pipe syntax: `collection | filter(predicate)`. Or it can be used with function call syntax: `filter(collection, predicate)` or `filter(predicate)(collection)`.

Example: `filter`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

void print(auto&& v)
{
    for (auto&& x : v)
```

```
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};
    auto myView = std::views::filter(v, [](int i) {return i < 5; });
    print(myView); // 0 1 2 3 -4

    auto myView2 = v | std::views::filter([](int i) {return i < 5; }); //
pipe syntax
    print(myView2); // 0 1 2 3 -4
}
```

Output

```
0 1 2 3 -4
0 1 2 3 -4
```

iota

Create a view that contains a sequence of increasing values. The sequence can be bounded or not.

C++

```
template<class V>
constexpr ranges::view auto iota(V&& startValue); // create an unbounded
sequence of incrementing values

template<class V, class E>
constexpr ranges::view auto iota(V&& startValue, E&& endValue); // create a
bounded sequence of incrementing values
```

Parameters

`E`

The type of the end value.

`S`

The type of the start value.

`startValue`

The first value in the sequence.

`endValue`

This value is one past the last value that will be in the sequence. For example, `std::views::iota(0, 5)` generates a view that has the values `0,1,2,3,4`.

Return value

An iota_view of a sequence of increasing values.

Remarks

For an unbounded sequence, the behavior is undefined after the maximum value of its datatype is reached.

Example: `iota`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>

void print(auto&& v)
{
    for (auto&& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    // create an iota view with its range adaptor (preferred)
    print(std::views::iota(0, 5)); // outputs 0 1 2 3 4

    // create an iota_view class directly
    std::ranges::iota_view letters{'a', 'f'};
    print(letters); // a b c d e
}
```

Output

```
0 1 2 3 4
a b c d e
```

`istream`

Create a view over the elements of a stream.

C++

```
template <class Val>
views::istream<Val>(str);
```

Parameters

`str`

A stream object. Its type is derived from a specialization of `std::basic_istream`.

`Val`

The type of the elements to extract from the stream.

Return value

A basic_istream_view.

This range adaptor is equivalent to `ranges::basic_istream_view<Val, typename U::char_type, typename U::traits_type>(str)`, where `U` is the type of `str`.

Example: `istream`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <sstream>
#include <vector>

int main()
{
    std::istringstream doubles{"1.1 2.2 3.3 4.4 5.5"};
    for (const auto& elem : std::views::istream<double>(doubles))
    {
        std::cout << elem << ' '; // 1.1 2.2 3.3 4.4 5.5
```

```
    }
}
```

Output

```
1.1 2.2 3.3 4.4 5.5
```

`join`

Create a view that combines all the elements of multiple ranges into a single view.

C++

```
1) template <ranges::viewable_range R>
[[nodiscard]] constexpr ranges::view auto join(R&& rg) const noexcept;

2) inline constexpr /*range adaptor closure*/ join();
```

Parameters

`R`

The type of the underlying range.

`rg`

The range to create the view from.

Return value

A join_view that contains the elements of all the ranges in the underlying range.

Example: `join`

C++

```
#include <iostream>
#include <vector>
#include <ranges>
#include <string>

int main()
{
    // a range of two ranges
    std::vector<std::string> rangeOfRanges[2]{{"C++", "20", "contains:"},
```

```
{"ranges", "modules", "concepts & more."}};

    for (const auto& elem : std::views::join(rangeOfRanges))
    {
        std::cout << elem << ' ';
    }
}
```

Output

```
C++ 20 contains: ranges modules concepts & more.
```

Remarks

The code shown earlier as "2)" can be used with pipe syntax: `collection | join`. Or it can be used with function call syntax: `join(collection)`.

keys

Create a keys_view of the first index into each tuple-like value in a collection. This is useful for extracting keys from associative containers. For example, given a range of `std::tuple<string, int>`, create a view that consists of all the `string` elements from each tuple.

C++

```
template <ranges::viewable_range R>
constexpr auto keys(R&& rg);
```

Parameters

`R`

The type of the underlying range.

Return value

A keys_view that consists of the first index into each tuple-like value in the range.

Example: keys

C++

```cpp
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <map>
#include <string>
#include <vector>

int main()
{
    // ========== extract keys from a map

    std::map<std::string, int> cpp_standards
    {
        {"C++98", 1998},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"C++20", 2020}
    };

    // Extract all of the keys from the map
    for (std::string standards : std::views::keys(cpp_standards))
    {
        std::cout << standards << ' '; // C++03 C++11 C++14 C++17 C++98
C++20
    }
    std::cout << '\n';

    // ========== Extract keys from a pair

    std::vector<std::pair<std::string, int>> windows
    {
        {"Windows 1.0", 1985},
        {"Windows 2.0", 1987},
        {"Windows 3.0", 1990},
        {"Windows 3.1", 1992},
        {"Windows NT 3.1", 1993},
        {"Windows 95", 1995},
        {"Windows NT 4.0", 1996},
        {"Windows 95", 1995},
        {"Windows 98", 1998},
        {"Windows 1.0", 1985},
        {"Windows 2000", 2000}
    };

    // Another way to call the range adaptor is by using '|'
    for (std::string version : windows | std::views::keys)
    {
        std::cout << version << ' '; // Windows 1.0 Windows 2.0 Windows 3.0
...
    }
}
```

Output

```
C++03 C++11 C++14 C++17 C++98 C++20
Windows 1.0 Windows 2.0 Windows 3.0 Windows 3.1 Windows NT 3.1 Windows 95
Windows NT 4.0 Windows 95 Windows 98 Windows 1.0 Windows 2000
```

`lazy_split`

Split a range into subranges based on a delimiter. The delimiter can be a single element or a view of elements.

C++

```
1) template<viewable_range R, class Pattern>
constexpr view auto lazy_split(R&& rg, Pattern&& delimiter);

2) template<class Pattern>
constexpr /*range adaptor closure*/ lazy_split(Pattern&& delimiter);
```

Parameters

`delimiter`
A single value, or a sequence of values that specify where to split the range.

`Pattern`
The type of the delimiter.

`R`
The type of the range to split.

`rg`
The range to split.

Return value

A lazy_split_view that contains one or more subranges and is the result of splitting the original range on `delimiter`.

Remarks

The delimiter isn't part of the result. For example, if you split the range `1,2,3` on the value `2`, you get two subranges: `1` and `3`.

A related adaptor is split. The primary differences between [`split_view](split-view-class.md) and lazy_split_view` are:

View	Can split a `const` range	Range iterator
`split_view`	no	Supports forward_range or higher
`lazy_split_view`	yes	input_range or higher

Prefer `split_view` because it's more efficient, unless you must split a range that is `const`.

The code shown earlier as "2)" can be used with pipe syntax: `collection | lazy_split(delimiter)`. Or it can be used with function call syntax: `lazy_split(collection, delimiter)` or `lazy_split(delimiter)(collection)`.

Example: `lazy_split`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> rg{1, 2, 3, 1, 2, 3, 4, 5, 6};

    // split on a single element
    for (const auto& sub : rg | std::views::split(3))
    {
        // outputs:
        // 1 2
        // 1 2
        // 4 5 6
        for (const auto& elem : sub)
        {
            std::cout << elem << ' ';
        }
        std::cout << '\n';
    }

    // split on a sequence of elements
    int delimiters[] = {2, 3};
    for (const auto& subrange : std::views::split(rg, delimiters))
    {
        // outputs 1 1 4 5 6
        for (auto& i : subrange)
```

```
        {
            std::cout << i << " ";
        }
    }
}
```

Output

```
1 2
1 2
4 5 6
1 1 4 5 6
```

`reverse`

Create a view of the elements of a range in reverse order.

C++

```
1) template<viewable_range R>
constexpr ranges::view auto reverse(R&& rg);

2) inline constexpr /*range adaptor closure*/ reverse();
```

Parameters

`R`

The type of the underlying range to reverse.

`rg`

The range to reverse.

Return value

A view that presents the elements of the underlying range in reverse order. The returned view is typically, but not always, a specialization of reverse_view. That is:

- If `V` is a specialization of `reverse_view`, the result is the argument's underlying view. A double-reverse is a no-op (no operation).
- If `V` has the form `subrange<reverse_iterator<I>, reverse_iterator<I>>`, the result is a subrange of the unwrapped iterators. A double-reverse is a no-op.
- Otherwise, the result is a reverse_view.

Remarks

The code shown earlier as "2)" can be used with pipe syntax: `collection | reverse`. Or it can be used with function call syntax: `reverse(collection)`.

Example: `reverse`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto rv = v | std::views::reverse; // using the pipe syntax

    for (auto &&e : rv) // outputs 6 5 -4 3 2 1 0
    {
        std::cout << e << ' ';
    }
    std::cout << '\n';

    // using the range adaptor without using the pipe syntax
    auto rv2 = std::views::reverse(v);
    for (auto &&e : rv2) // outputs 6 5 -4 3 2 1 0
    {
        std::cout << e << ' ';
    }
}
```

Output

```
6 5 -4 3 2 1 0
6 5 -4 3 2 1 0
```

`single`

Create a `single_view`, which is a view that contains one element.

C++

```
template<class T>
constexpr ranges::view auto single(T&& t);
```

Parameters

`T`

The type of the element in the view.

`t`

The value of the element to store in the view.

Return value

An single_view that contains `t`.

Remarks

This view is useful for test purposes, for calling code that needs to be provided with a view that has at least one element in it.

Example: `single`

C++

```
// requires /std:c++20 or higher
#include <ranges>
#include <string>
#include <tuple>
#include <iostream>

int main()
{
    auto sv = std::views::single(7);
    std::cout << sv.front() << " " << *sv.data() << "\n"; // 7 7

    auto sv2 = std::views::single(<std::tuple<double, std::string>{6502, "8-
bit"});
    std::cout << std::get<0>(sv2[0]) << " " << std::get<1>(sv2[0]) << "\n";
// 6502 8-bit
}
```

Output

```
7 7
6502 8-bit
```

`split`

Split a view into subranges based on a delimiter. The delimiter can be a single element or a sequence of elements.

```
C++
```

```
1) template<viewable_range R, class Pattern>
constexpr view auto split(R&& rg, Pattern&& delimiter);

2) template<class Pattern>
constexpr /*range adaptor closure*/ split(Pattern&& delimiter);
```

Parameters

`delimiter`

A single value, or a sequence of values that specify where to split the range.

`Pattern`

The type of the delimiter.

`R`

The type of the underlying range to split.

`rg`

The range to split.

Return value

A split_view that contains one or more subranges.

Remarks

The delimiter isn't part of the result. For example, if you split the range `1,2,3` on the value `2`, you get two subranges: `1` and `3`.

A related adaptor is lazy_split. The primary differences between `split_view` and `lazy_split_view` are:

View	Can split a `const` range	Range type
`split_view`	no	Supports forward_range or higher
`lazy_split_view`	yes	Supports input_range or higher

Prefer `split_view` because it's more efficient, unless you must split a range that is `const`.

The code shown earlier as "2)" can be used with pipe syntax: `collection | split(delimiter)`. Or it can be used with function call syntax: `split(collection, 5)` or `split(5)(collection)`.

Example: `split`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> rg{ 1, 2, 3, 1, 2, 3, 4, 5, 6 };

    // split on a single element, 3
    for (const auto& sub : rg | std::views::split(3))
    {
        // This prints out:
        // 1,2
        // 4,5,6
        for (const auto& elem : sub)
        {
            std::cout << elem << ' ';
        }
        std::cout << '\n';
    }

    // split on a sequence of elements, 2,3
    int delimiters[] = {2, 3};
    for (const auto& subrange : std::views::split(rg, delimiters))
    {
        // outputs 1 1 4 5 6
        for (auto& i : subrange)
        {
            std::cout << i << " ";
        }
    }
}
```

Output

```
1 2
1 2
```

```
4 5 6
1 1 4 5 6
```

`take`

Create a view that contains the specified number of elements taken from the front of a range.

C++

```
1) template<ranges::viewable_range R>
constexpr ranges::view auto take(R&& rg, ranges::range_difference_type<R>
count);

2) template<class DifferenceType>
constexpr /*range adaptor closure*/ take(DifferenceType&& count);
```

Parameters

`R`
The type of the underlying range.

`rg`
The range to create the view from.

`count`
The number of elements to take from the front of `rg`.

Return value

The returned view is typically, but not always, a specialization of take_view. Specifically:

- If `V` is a specialization of empty_view, or is a specialization of span, basic_string_view, iota_view, or subrange that is both random_access_range and sized_range, the result is a specialization of `V`.
- Otherwise, the result is a take_view.

Remarks

If you specify more elements to take than exist in `rg`, all of the elements are taken.

`take` is the opposite of drop.

The code shown earlier as "2)" can be used with pipe syntax: `collection | take(5)`. Or it can be used with function call syntax: `take(5, collection)` or `take(5)(collection)`.

Example: `take`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::string s{ "abcdefg" };
    auto myView = std::views::take(s, 3);
    for (auto c : myView)
    {
        std::cout << c << ' '; // a b c
    }

    std::cout << std::endl;

    for (auto c : s | std::views::take(3)) // pipe syntax
    {
        std::cout << c << ' '; // a b c
    }
}
```

Output

```
a b c
a b c
```

`take_while`

Create a view that contains the leading elements of a range that match the specified condition.

C++

```
1) template<ranges::viewable_range R, class P>
constexpr ranges::view auto take_while(R&& rg, P&& predicate);

2) template<class P>
constexpr /*range adaptor closure*/ take_while(P&& predicate);
```

Parameters

`P`

The type of the predicate.

`predicate`

The conditions that determine which leading elements to copy from the range.

`R`

The type of the underlying range.

`rg`

The range to create the view from.

Return value

A take_while_view that consists of the first `count` elements that meet the specified criteria in the range.

Remarks

Stops taking elements from `rg` after the predicate returns `false` or the range runs out of elements.

`take_while` is the opposite of drop_while.

The code shown earlier as "2)" can be used with pipe syntax: `collection | take_while(pred)`. Or it can be used with function call syntax: `take_while(collection, pred)` or `take_while(pred)(collection)`.

Example: `take_while`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

void print(auto&& v)
{
    for (auto&& x : v)
```

```
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{ 0, 1, 2, 3, -4, 5, 6 };
    auto myView = std::views::take_while(
        v, [](int i) {return i >= 0; });
    print(myView); // 0 1 2 3

    print(v | std::views::take_while( // 0 1 2 3 -4
        [](int i) {return i < 5; })); // pipe syntax
}
```

Output

```
0 1 2 3
0 1 2 3 -4
```

`transform`

Create a view of elements, each of which is a transformation of an element in the specified range.

C++

```
1) template<viewable_range R, class F>
constexpr ranges::view auto transform(R&& rg, F&& fun);

2) template<class F>
constexpr /*range adaptor closure*/ transform(F&& fun);
```

Parameters

`F`

The type of the function object to transform the elements.

`R`

The type of the underlying range.

`fun`

The function that transforms the elements.

`rg`

The range to create the view from.

Return value

A transform_view that contains the transformed elements of `rg`.

Remarks

For efficiency's sake, when you compose `filter` and `transform`, do the `filter` first so that you `transform` only the elements that you intend to keep.

The code shown earlier as "2)" can be used with pipe syntax: `collection | transform(fun)`. Or it can be used with function call syntax: `transform(collection, fun)` or `transform(fun)(collection)`.

Example: `transform`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <vector>

void print(auto&& v)
{
    for (auto&& x : v)
    {
        std::cout << x << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{0, 1, 2, 3, -4, 5, 6};
    auto myView = std::views::transform(v, [](int i) {return i * 2; });
    print(myView); // 0 2 4 6 -8 10 12

    print(v | std::views::transform( // 0 2 4 6 -8 10 12
        [](int i) {return i * 2; })); // pipe syntax
}
```

Output

```
0 2 4 6 -8 10 12
0 2 4 6 -8 10 12
```

`values`

Create a values_view that consists of the second index into each tuple-like value in a collection. This is useful for making a view of the values in an associative container. For example, given a range of `std::tuple<string, int>` values, create a view that consists of all the `int` elements from each tuple.

C++

```
template <range::viewable_range R>
constexpr ranges::view auto values(R&& rg);
```

Parameters

`R`

The type of the underlying range.

`rg`

The underlying range of tuple-like values.

Return value

A values_view built from the second index into each tuple-like value in the range.

Example: `values`

C++

```
// requires /std:c++20 or later
#include <ranges>
#include <iostream>
#include <map>
#include <string>
#include <vector>

int main()
{
    // ========== working with a map

    std::map<std::string, int> cpp_standards
```

```
    {
        {"C++98", 1998},
        {"C++03", 2003},
        {"C++11", 2011},
        {"C++14", 2014},
        {"C++17", 2017},
        {"C++20", 2020}
    };

    // Extract all of the years from the map
    for (int years : std::views::values(cpp_standards))
    {
        std::cout << years << ' '; // 2003 2011 2014 2017 1998 2020
    }
    std::cout << '\n';

    // ========== working with pairs

    std::vector<std::pair<std::string, int>> windows
    {
        {"Windows 1.0", 1985},
        {"Windows 2.0", 1987},
        {"Windows 3.0", 1990},
        {"Windows 3.1", 1992},
        {"Windows NT 3.1", 1993},
        {"Windows 95", 1995},
        {"Windows NT 4.0", 1996},
        {"Windows 95", 1995},
        {"Windows 98", 1998},
        {"Windows 1.0", 1985},
        {"Windows 2000", 2000}
    };

    // Another way to call the range adaptor by using '|'
    // Create a values_view that contains the year from each pair
    for (int years : windows | std::views::values)
    {
        std::cout << years << ' '; // 1985 1987 1990 1992 ...
    }
}
```

Output

```
2003 2011 2014 2017 1998 2020
1985 1987 1990 1992 1993 1995 1996 1995 1998 1985 2000
```

Range adaptor type aliases

`all_t`

Provides the type of the view that all returns.

```
C++
```

```
template <ranges::viewable_range R>
using all_t = decltype(views::all(std::declval<R>()));
```

Parameters

`R`

The type of the underlying range.

Return value

The type of the view that `all` returns: `decltype(views::all(std::declval<R>()))`.

Example: `all_t`

```
C++
```

```
#include <ranges>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {1,2,3,4,5,6,7,8,9,10};
    auto myView = std::views::all(v);
    std::views::all_t<decltype((v))> &viewType = myView;
}
```

See also

<ranges>
<ranges> concepts
View classes

<ratio>

Article • 12/07/2021

Include the standard header <ratio> to define constants and templates that are used to store and manipulate rational numbers at compile time.

Syntax

C++

```
#include <ratio>
```

ratio Template

C++

```
template<std::intmax_t Numerator, std::intmax_t Denominator = 1>
struct ratio // holds the ratio of Numerator to Denominator
{
   static constexpr std::intmax_t num;
   static constexpr std::intmax_t den;
   typedef ratio<num, den> type;
}
```

The template `ratio` defines the static constants `num` and `den` such that `num` / `den` == Numerator / Denominator and `num` and `den` have no common factors. `num` / `den` is the value that is represented by the class template. Therefore, `type` designates the instantiation `ratio<num, den>`.

Specializations

<ratio> also defines specializations of `ratio` that have the following form.

`template <class R1, class R2> struct ratio_specialization`

Each specialization takes two template parameters that must also be specializations of `ratio`. The value of `type` is determined by an associated logical operation.

Name	`type` Value
`ratio_add`	`R1 + R2`

Name	`type` Value
`ratio_divide`	`R1 / R2`
`ratio_equal`	`R1 == R2`
`ratio_greater`	`R1 > R2`
`ratio_greater_equal`	`R1 >= R2`
`ratio_less`	`R1 < R2`
`ratio_less_equal`	`R1 <= R2`
`ratio_multiply`	`R1 * R2`
`ratio_not_equal`	`!(R1 == R2)`
`ratio_subtract`	`R1 - R2`

typedefs

For convenience, the header defines ratios for the standard SI prefixes:

C++

```
typedef ratio<1, 1000000000000000000> atto;
typedef ratio<1, 1000000000000000> femto;
typedef ratio<1, 1000000000000> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000, 1> tera;
typedef ratio<1000000000000000, 1> peta;
typedef ratio<1000000000000000000, 1> exa;
```

See also

Header Files Reference

<regex>

Article • 02/25/2022

Defines a class template to parse Regular Expressions (C++), and several class templates and functions to search text for matches to a regular expression object.

Syntax

```
C++

#include <regex>
```

Remarks

To create a regular expression object, use the class template basic_regex Class or one of its specializations, regex and wregex, together with the syntax flags of type regex_constants::syntax_option_type.

To search text for matches to a regular expression object, use the template functions regex_match and regex_search, together with the match flags of type regex_constants::match_flag_type. These functions return results by using the class template match_results Class and its specializations, cmatch, wcmatch, smatch, and wsmatch, together with the class template sub_match Class and its specializations, csub_match, wcsub_match, ssub_match, and wssub_match.

To replace text that matches a regular expression object, use the template function regex_replace, together with the match flags of type regex_constants::match_flag_type.

To iterate through multiple matches of a regular expression object, use the class templates regex_iterator Class and regex_token_iterator Class or one of their specializations, cregex_iterator, sregex_iterator, wcregex_iterator, wsregex_iterator, cregex_token_iterator, sregex_token_iterator, wcregex_token_iterator, or wsregex_token_iterator, together with the match flags of type regex_constants::match_flag_type.

To modify the details of the grammar of regular expressions, write a class that implements the regular expression traits.

Classes

Class	Description
basic_regex	Wraps a regular expression.
match_results	Holds a sequence of submatches.
regex_constants	Holds assorted constants.
regex_error	Reports a bad regular expression.
regex_iterator	Iterates through match results.
regex_traits	Describes characteristics of elements for matching.
regex_traits<char>	Describes characteristics of `char` for matching.
regex_traits<wchar_t>	Describes characteristics of `wchar_t` for matching.
regex_token_iterator	Iterates through submatches.
sub_match	Describes a submatch.

Type Definitions

Name	Description
cmatch	Type definition for `char` `match_results`.
cregex_iterator	Type definition for `char` `regex_iterator`.
cregex_token_iterator	Type definition for `char` `regex_token_iterator`.
csub_match	Type definition for `char` `sub_match`.
regex	Type definition for `char` `basic_regex`.
smatch	Type definition for `string` `match_results`.
sregex_iterator	Type definition for `string` `regex_iterator`.
sregex_token_iterator	Type definition for `string` `regex_token_iterator`.
ssub_match	Type definition for `string` `sub_match`.
wcmatch	Type definition for `wchar_t` `match_results`.
wcregex_iterator	Type definition for `wchar_t` `regex_iterator`.
wcregex_token_iterator	Type definition for `wchar_t` `regex_token_iterator`.
wcsub_match	Type definition for `wchar_t` `sub_match`.

Name	Description
wregex	Type definition for `wchar_t` `basic_regex`.
wsmatch	Type definition for `wstring` `match_results`.
wsregex_iterator	Type definition for `wstring` `regex_iterator`.
wsregex_token_iterator	Type definition for `wstring` `regex_token_iterator`.
wssub_match	Type definition for `wstring` `sub_match`.

Functions

Function	Description
regex_match	Exactly matches a regular expression.
regex_replace	Replaces matched regular expressions.
regex_search	Searches for a regular expression match.
swap	Swaps `basic_regex` or `match_results` objects.

Operators

Operator	Description
operator==	Comparison of various objects, equal.
operator!=	Comparison of various objects, not equal.
operator<	Comparison of various objects, less than.
operator\<=	Comparison of various objects, less than or equal.
operator>	Comparison of various objects, greater than.
operator>=	Comparison of various objects, greater than or equal.
operator<<	Inserts a `sub_match` in a stream.

See also

Regular Expressions (C++)
regex_constants Class

regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

`<regex>` functions

Article • 02/08/2022

Name	Description
regex_match	Tests whether a regular expression matches the entire target string.
regex_replace	Replaces matched regular expressions.
regex_search	Searches for a regular expression match.
swap	Swaps two `basic_regex` or `match_results` objects.

`regex_match`

Tests whether a regular expression matches the entire target string.

C++

```
// (1)
template <class BidIt, class Alloc, class Elem, class RXtraits, class
Alloc2>
bool regex_match(
    BidIt first,
    Bidit last,
    match_results<BidIt, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (2)
template <class BidIt, class Elem, class RXtraits, class Alloc2>
bool regex_match(
    BidIt first,
    Bidit last,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (3)
template <class Elem, class Alloc, class RXtraits, class Alloc2>
bool regex_match(
    const Elem *ptr,
    match_results<const Elem*, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (4)
template <class Elem, class RXtraits, class Alloc2>
bool regex_match(
    const Elem *ptr,
```

```
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (5)
template <class IOtraits, class IOalloc, class Alloc, class Elem, class
RXtraits, class Alloc2>
bool regex_match(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    match_results<typename basic_string<Elem, IOtraits,
IOalloc>::const_iterator, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

// (6)
template <class IOtraits, class IOalloc, class Elem, class RXtraits, class
Alloc2>
bool regex_match(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);
```

Parameters

`BidIt`

The iterator type for submatches. For common cases this one of `string::const_iterator`, `wstring::const_iterator`, `const char*` or `const wchar_t*`.

`Alloc`

The match results allocator class.

`Elem`

The type of elements to match. For common cases this is `string`, `wstring`, `char*` or `wchar_t*`.

`RXtraits`

Traits class for elements.

`Alloc2`

The regular expression allocator class.

`IOtraits`

The string traits class.

`IOalloc`

The string allocator class.

flags

Flags for matches.

first

Beginning of sequence to match.

last

End of sequence to match.

match

The match results. Corresponds to Elem type: smatch for `string`, wsmatch for `wstring`, cmatch for `char*` or wcmatch for `wchar_t*`.

ptr

Pointer to beginning of sequence to match. If *ptr* is `char*`, then use `cmatch` and `regex`. If *ptr* is `wchar_t*` then use `wcmatch` and `wregex`.

re

The regular expression to match. Type `regex` for `string` and `char*`, or `wregex` for `wstring` and `wchar_t*`.

str

String to match. Corresponds to the type of *Elem*.

Remarks

Each template function returns true only if the entire operand sequence *str* exactly matches the regular expression argument *re*. Use regex_search to match a substring within a target sequence and `regex_iterator` to find multiple matches. The functions that take a `match_results` object set its members to reflect whether the match succeeded and if so what the various capture groups in the regular expression captured.

The functions that take a `match_results` object set its members to reflect whether the match succeeded and if so what the various capture groups in the regular expression captured.

Example

C++

```
// std__regex__regex_match.cpp
// compile with: /EHsc
```

```
#include <regex>
#include <iostream>

using namespace std;

int main()
{
    // (1) with char*
    // Note how const char* requires cmatch and regex
    const char *first = "abc";
    const char *last = first + strlen(first);
    cmatch narrowMatch;
    regex rx("a(b)c");

    bool found = regex_match(first, last, narrowMatch, rx);
    if (found)
        wcout << L"Regex found in abc" << endl;

    // (2) with std::wstring
    // Note how wstring requires wsmatch and wregex.
    // Note use of const iterators cbegin() and cend().
    wstring target(L"Hello");
    wsmatch wideMatch;
    wregex wrx(L"He(l+)o");

    if (regex_match(target.cbegin(), target.cend(), wideMatch, wrx))
        wcout << L"The matching text is:" << wideMatch.str() << endl;

    // (3) with std::string
    string target2("Drizzle");
    regex rx2(R"(D\w+e)"); // no double backslashes with raw string literal

    found = regex_match(target2.cbegin(), target2.cend(), rx2);
    if (found)
        wcout << L"Regex found in Drizzle" << endl;

    // (4) with wchar_t*
    const wchar_t* target3 = L"2014-04-02";
    wcmatch wideMatch2;

    // LR"(...)" is a  raw wide-string literal. Open and close parens
    // are delimiters, not string elements.
    wregex wrx2(LR"(\d{4}(-|/)\d{2}(-|/)\d{2})");
    if (regex_match(target3, wideMatch2, wrx2))
    {
        wcout << L"Matching text: " << wideMatch2.str() << endl;
    }

     return 0;
}
```

Output

```
Regex found in abc
The matching text is: Hello
Regex found in Drizzle
The matching text is: 2014-04-02
```

`regex_replace`

Replaces matched regular expressions.

C++

```
template <class OutIt, class BidIt, class RXtraits, class Alloc, class Elem>
OutIt regex_replace(
    OutIt out,
    BidIt first,
    BidIt last,
    const basic_regex<Elem, RXtraits, Alloc>& re,
    const basic_string<Elem>& fmt,
    match_flag_type flags = match_default);

template <class RXtraits, class Alloc, class Elem>
basic_string<Elem> regex_replace(
    const basic_string<Elem>& str,
    const basic_regex<Elem, RXtraits, Alloc>& re,
    const basic_string<Elem>& fmt,
    match_flag_type flags = match_default);
```

Parameters

`OutIt`

The iterator type for replacements.

`BidIt`

The iterator type for submatches.

`RXtraits`

Traits class for elements.

`Alloc`

The regular expression allocator class.

`Elem`

The type of elements to match.

flags

Flags for matches.

first

Beginning of sequence to match.

fmt

The format for replacements.

last

End of sequence to match.

out

The output iterator.

re

The regular expression to match.

str

String to match.

Remarks

The first function constructs a regex_iterator Class object `iter(first, last, re, flags)` and uses it to split its input range `[first, last)` into a series of subsequences `T0 M0 T1 M1...TN-1 MN-1 TN`, where `Mn` is the nth match detected by the iterator. If no matches are found, `T0` is the entire input range and `N` is zero. If `(flags & format_first_only) != 0` only the first match is used, `T1` is all of the input text that follows the match, and `N` is 1. For each `i` in the range `[0, N)`, if `(flags & format_no_copy) == 0` it copies the text in the range `Ti` to the iterator *out*. It then calls `m.format(out, fmt, flags)`, where `m` is the `match_results` object returned by the iterator object `iter` for the subsequence `Mi`. Finally, if `(flags & format_no_copy) == 0` it copies the text in the range `TN` to the iterator *out*. The function returns *out*.

The second function constructs a local variable `result` of type `basic_string<charT>` and calls `regex_replace(back_inserter(result), str.begin(), str.end(), re, fmt, flags)`. It returns `result`.

Example

```
C++
```

```cpp
// std__regex__regex_replace.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    char buf[20];
    const char *first = "axayaz";
    const char *last = first + strlen(first);
    std::regex rx("a");
    std::string fmt("A");
    std::regex_constants::match_flag_type fonly =
        std::regex_constants::format_first_only;

    *std::regex_replace(&buf[0], first, last, rx, fmt) = '\0';
    std::cout << "replacement == " << &buf[0] << std::endl;

    *std::regex_replace(&buf[0], first, last, rx, fmt, fonly) = '\0';
    std::cout << "replacement == " << &buf[0] << std::endl;

    std::string str("adaeaf");
    std::cout << "replacement == "
        << std::regex_replace(str, rx, fmt) << std::endl;

    std::cout << "replacement == "
        << std::regex_replace(str, rx, fmt, fonly) << std::endl;

    return (0);
}
```

Output

```
replacement == AxAyAz
replacement == Axayaz
replacement == AdAeAf
replacement == Adaeaf
```

regex_search

Searches for a regular expression match.

C++

```cpp
template <class BidIt, class Alloc, class Elem, class RXtraits, class
Alloc2>
bool regex_search(
    BidIt first,
    Bidit last,
    match_results<BidIt, Alloc>& match,
```

```
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class BidIt, class Elem, class RXtraits, class Alloc2>
bool regex_search(
    BidIt first,
    Bidit last,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class Elem, class Alloc, class RXtraits, class Alloc2>
bool regex_search(
    const Elem* ptr,
    match_results<const Elem*, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class Elem, class RXtraits, class Alloc2>
bool regex_search(
    const Elem* ptr,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class IOtraits, class IOalloc, class Alloc, class Elem, class
RXtraits, class Alloc2>
bool regex_search(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    match_results<typename basic_string<Elem, IOtraits,
IOalloc>::const_iterator, Alloc>& match,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);

template <class IOtraits, class IOalloc, class Elem, class RXtraits, class
Alloc2>
bool regex_search(
    const basic_string<Elem, IOtraits, IOalloc>& str,
    const basic_regex<Elem, RXtraits, Alloc2>& re,
    match_flag_type flags = match_default);
```

Parameters

BidIt

The iterator type for submatches.

Alloc

The match results allocator class.

Elem

The type of elements to match.

RXtraits

Traits class for elements.

Alloc2

The regular expression allocator class.

IOtraits

The string traits class.

IOalloc

The string allocator class.

flags

Flags for matches.

first

Beginning of sequence to match.

last

End of sequence to match.

match

The match results.

ptr

Pointer to beginning of sequence to match.

re

The regular expression to match.

str

String to match.

Remarks

Each template function returns true only if a search for its regular expression argument *re* in its operand sequence succeeds. The functions that take a `match_results` object set its members to reflect whether the search succeeded and if so what the various capture groups in the regular expression captured.

Example

```
C++
```

```cpp
// std__regex__regex_search.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    const char *first = "abcd";
    const char *last = first + strlen(first);
    std::cmatch mr;
    std::regex rx("abc");
    std::regex_constants::match_flag_type fl =
        std::regex_constants::match_default;

    std::cout << "search(f, f+1, \"abc\") == " << std::boolalpha
        << regex_search(first, first + 1, rx, fl) << std::endl;

    std::cout << "search(f, l, \"abc\") == " << std::boolalpha
        << regex_search(first, last, mr, rx) << std::endl;
    std::cout << "  matched: \"" << mr.str() << "\"" << std::endl;

    std::cout << "search(\"a\", \"abc\") == " << std::boolalpha
        << regex_search("a", rx) << std::endl;

    std::cout << "search(\"xabcd\", \"abc\") == " << std::boolalpha
        << regex_search("xabcd", mr, rx) << std::endl;
    std::cout << "  matched: \"" << mr.str() << "\"" << std::endl;

    std::cout << "search(string, \"abc\") == " << std::boolalpha
        << regex_search(std::string("a"), rx) << std::endl;

    std::string str("abcabc");
    std::match_results<std::string::const_iterator> mr2;
    std::cout << "search(string, \"abc\") == " << std::boolalpha
        << regex_search(str, mr2, rx) << std::endl;
    std::cout << "  matched: \"" << mr2.str() << "\"" << std::endl;

    return (0);
}
```

Output

```
search(f, f+1, "abc") == false
search(f, l, "abc") == true
  matched: "abc"
search("a", "abc") == false
search("xabcd", "abc") == true
  matched: "abc"
search(string, "abc") == false
search(string, "abc") == true
  matched: "abc"
```

swap

Swaps two `basic_regex` or `match_results` objects.

```
C++
```

```
template <class Elem, class RXtraits>
void swap(
    basic_regex<Elem, RXtraits, Alloc>& left,
    basic_regex<Elem, RXtraits>& right) noexcept;

template <class Elem, class IOtraits, class BidIt, class Alloc>
void swap(
    match_results<BidIt, Alloc>& left,
    match_results<BidIt, Alloc>& right) noexcept;
```

Parameters

`Elem`

The type of elements to match.

`RXtraits`

Traits class for elements.

Remarks

The template functions swap the contents of their respective arguments in constant time and don't throw exceptions.

Example

```
C++
```

```
// std__regex__swap.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx0("c(a*)|(b)");
    std::regex rx1;
    std::cmatch mr0;
    std::cmatch mr1;

    swap(rx0, rx1);
```

```
    std::regex_search("xcaaay", mr1, rx1);
    swap(mr0, mr1);

    std::csub_match sub = mr0[1];
    std::cout << "matched == " << std::boolalpha
        << sub.matched << std::endl;
    std::cout << "length == " << sub.length() << std::endl;
    std::cout << "string == " << sub << std::endl;

    return (0);
}
```

Output

```
matched == true
length == 3
string == aaa
```

See also

<regex>
regex_constants Class
regex_error Class
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

`<regex>` operators

Article • 12/07/2021

operator!=
operator>
operator>=
operator<
operator<<
operator<=
operator==

operator!=

Not equal comparison for various objects.

C++

```
template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator!=(
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator!=(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator!=(const typename iterator_traits<BidIt>::value_type *left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator!=(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator!=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);
```

```
template <class BidIt, class Alloc>
bool operator!=(const match_results<BidIt, Alloc>& left,
    const match_results<BidIt, Alloc>& right);
```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `!(left == right)`.

Example

C++

```
// std__regex__operator_ne.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "match == " << mr.str() << std::endl;
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;
```

```
    std::cout << "match != match == " << std::boolalpha
        << (mr != mr) << std::endl;
    std::cout << "sub != sub == " << std::boolalpha
        << (sub != sub) << std::endl;

    std::cout << "string(\"aab\") != sub == " << std::boolalpha
        << (Mystr("aab") != sub) << std::endl;
    std::cout << "sub != string(\"aab\") == " << std::boolalpha
        << (sub != Mystr("aab")) << std::endl;

    std::cout << "\"aab\" != sub == " << std::boolalpha
        << ("aab" != sub) << std::endl;
    std::cout << "sub != \"aab\" == " << std::boolalpha
        << (sub != "aab") << std::endl;

    std::cout << "'a' != sub == " << std::boolalpha
        << ('a' != sub) << std::endl;
    std::cout << "sub != 'a' == " << std::boolalpha
        << (sub != 'a') << std::endl;

    return (0);
    }
```

Output

```
match == caaa
sub == aaa

match != match == false
sub != sub == false
string("aab") != sub == true
sub != string("aab") == true
"aab" != sub == true
sub != "aab" == true
'a' != sub == true
sub != 'a' == true
```

operator<

Less than comparison for various objects.

C++

```
template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<(
```

```
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator<(const typename iterator_traits<BidIt>::value_type *left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator<(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator<(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);
```

Parameters

BidIt
The iterator type.

IOtraits
The string traits class.

Alloc
The allocator class.

left
The left object to compare.

right
The right object to compare.

Remarks

Each template operator converts its arguments to a string type and returns true only if the converted value of *left* compares less than the converted value of *right*.

Example

C++

```
// std__regex__operator_lt.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub < sub == " << std::boolalpha
        << (sub < sub) << std::endl;

    std::cout << "string(\"aab\") < sub == " << std::boolalpha
        << (Mystr("aab") < sub) << std::endl;
    std::cout << "sub < string(\"aab\") == " << std::boolalpha
        << (sub < Mystr("aab")) << std::endl;

    std::cout << "\"aab\" < sub == " << std::boolalpha
        << ("aab" < sub) << std::endl;
    std::cout << "sub < \"aab\" == " << std::boolalpha
        << (sub < "aab") << std::endl;

    std::cout << "'a' < sub == " << std::boolalpha
        << ('a' < sub) << std::endl;
    std::cout << "sub < 'a' == " << std::boolalpha
        << (sub < 'a') << std::endl;

    return (0);
    }
```

Output

```
sub == aaa

sub < sub == false
string("aab") < sub == false
sub < string("aab") == true
"aab" < sub == false
sub < "aab" == true
```

```
'a' < sub == true
sub < 'a' == false
```

`operator<<`

Inserts a sub_match in a stream.

C++

```
template <class Elem, class IOtraits, class Alloc, class BidIt>
basic_ostream<Elem, IOtraits>& operator<<(basic_ostream<Elem, IOtraits>& os,
    const sub_match<BidIt>& right);
```

Parameters

Elem
The element type.

IOtraits
The string traits class.

Alloc
The allocator class.

BidIt
The iterator type.

os
The output stream.

right
The object to insert.

Remarks

The template operator returns `os << right.str()`.

Example

C++

```
// std__regex__operator_ins.cpp
// compile with: /EHsc
```

```
#include <regex>
#include <iostream>

int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[0];
    std::cout << "whole match: " << sub << std::endl;

    return (0);
    }
```

Output

```
whole match: caaa
```

operator<=

Less than or equal comparison for various objects.

C++

```
template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<=(
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator<=(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator<=(const typename iterator_traits<BidIt>::value_type *left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
```

```
bool operator<=(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator<=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);
```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `!(right < left)`.

Example

C++

```
// std__regex__operator_le.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
```

```
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub <= sub == " << std::boolalpha
        << (sub <= sub) << std::endl;

    std::cout << "string(\"aab\") <= sub == " << std::boolalpha
        << (Mystr("aab") <= sub) << std::endl;
    std::cout << "sub <= string(\"aab\") == " << std::boolalpha
        << (sub <= Mystr("aab")) << std::endl;

    std::cout << "\"aab\" <= sub == " << std::boolalpha
        << ("aab" <= sub) << std::endl;
    std::cout << "sub <= \"aab\" == " << std::boolalpha
        << (sub <= "aab") << std::endl;

    std::cout << "'a' <= sub == " << std::boolalpha
        << ('a' <= sub) << std::endl;
    std::cout << "sub <= 'a' == " << std::boolalpha
        << (sub <= 'a') << std::endl;

    return (0);
    }
```

Output

```
sub == aaa

sub <= sub == true
string("aab") <= sub == false
sub <= string("aab") == true
"aab" <= sub == false
sub <= "aab" == true
'a' <= sub == true
sub <= 'a' == false
```

operator==

Equal comparison for various objects.

C++

```
template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator==(
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& left,
```

```
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator==(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator==(const typename iterator_traits<BidIt>::value_type* left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type* right);

template <class BidIt>
bool operator==(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator==(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);

template <class BidIt, class Alloc>
bool operator==(const match_results<BidIt, Alloc>& left,
    const match_results<BidIt, Alloc>& right);
```

Parameters

BidIt
The iterator type.

IOtraits
The string traits class.

Alloc
The allocator class.

left
The left object to compare.

right
The right object to compare.

Remarks

Each template operator converts each of its arguments to a string type and returns the result of comparing the converted objects for equality.

When a template operator converts its arguments to a string type it uses the first of the following transformations that applies:

arguments whose types are a specialization of class template `match_results` or `sub_match` are converted by calling the `str` member function;

arguments whose types are a specialization of the class template `basic_string` are unchanged;

all other argument types are converted by passing the argument value to the constructor for an appropriate specialization of the class template `basic_string`.

Example

C++

```
// std__regex__operator_eq.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "match == " << mr.str() << std::endl;
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "match == match == " << std::boolalpha
        << (mr == mr) << std::endl;
    std::cout << "sub == sub == " << std::boolalpha
        << (sub == sub) << std::endl;

    std::cout << "string(\"aab\") == sub == " << std::boolalpha
        << (Mystr("aab") == sub) << std::endl;
    std::cout << "sub == string(\"aab\") == " << std::boolalpha
        << (sub == Mystr("aab")) << std::endl;

    std::cout << "\"aab\" == sub == " << std::boolalpha
        << ("aab" == sub) << std::endl;
    std::cout << "sub == \"aab\" == " << std::boolalpha
        << (sub == "aab") << std::endl;

    std::cout << "'a' == sub == " << std::boolalpha
```

```
        << ('a' == sub) << std::endl;
    std::cout << "sub == 'a' == " << std::boolalpha
        << (sub == 'a') << std::endl;

    return (0);
    }
```

Output

```
match == caaa
sub == aaa

match == match == true
sub == sub == true
string("aab") == sub == false
sub == string("aab") == false
"aab" == sub == false
sub == "aab" == false
'a' == sub == false
sub == 'a' == false
```

operator>

Greater than comparison for various objects.

C++

```
template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>(
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator>(const typename iterator_traits<BidIt>::value_type *left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type *right);
```

```
template <class BidIt>
bool operator>(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator>(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);
```

Parameters

BidIt

The iterator type.

IOtraits

The string traits class.

Alloc

The allocator class.

left

The left object to compare.

right

The right object to compare.

Remarks

Each template operator returns `right < left`.

Example

C++

```
// std__regex__operator_gt.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);
```

```
    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub > sub == " << std::boolalpha
        << (sub > sub) << std::endl;

    std::cout << "string(\"aab\") > sub == " << std::boolalpha
        << (Mystr("aab") > sub) << std::endl;
    std::cout << "sub > string(\"aab\") == " << std::boolalpha
        << (sub > Mystr("aab")) << std::endl;

    std::cout << "\"aab\" > sub == " << std::boolalpha
        << ("aab" > sub) << std::endl;
    std::cout << "sub > \"aab\" == " << std::boolalpha
        << (sub > "aab") << std::endl;

    std::cout << "'a' > sub == " << std::boolalpha
        << ('a' > sub) << std::endl;
    std::cout << "sub > 'a' == " << std::boolalpha
        << (sub > 'a') << std::endl;

    return (0);
    }
```

Output

```
sub == aaa

sub > sub == false
string("aab") > sub == true
sub > string("aab") == false
"aab" > sub == true
sub > "aab" == false
'a' > sub == false
sub > 'a' == true
```

operator>=

Greater than or equal comparison for various objects.

C++

```
template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>=(
    const basic_string<typename iterator_traits<BidIt>::value_type,
```

```
IOtraits, Alloc>& left,
    const sub_match<BidIt>& right);

template <class BidIt, class IOtraits, class Alloc>
bool operator>=(const sub_match<BidIt>& left,
    const basic_string<typename iterator_traits<BidIt>::value_type,
IOtraits, Alloc>& right);

template <class BidIt>
bool operator>=(const typename iterator_traits<BidIt>::value_type *left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type *right);

template <class BidIt>
bool operator>=(const typename iterator_traits<BidIt>::value_type& left,
    const sub_match<BidIt>& right);

template <class BidIt>
bool operator>=(const sub_match<BidIt>& left,
    const typename iterator_traits<BidIt>::value_type& right);
```

Parameters

BidIt
The iterator type.

IOtraits
The string traits class.

Alloc
The allocator class.

left
The left object to compare.

right
The right object to compare.

Remarks

Each template operator returns `!(left < right)`.

Example

C++

```cpp
// std__regex__operator_ge.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::cmatch::string_type Mystr;
int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "sub == " << sub << std::endl;
    std::cout << std::endl;

    std::cout << "sub >= sub == " << std::boolalpha
        << (sub >= sub) << std::endl;

    std::cout << "string(\"aab\") >= sub == " << std::boolalpha
        << (Mystr("aab") >= sub) << std::endl;
    std::cout << "sub >= string(\"aab\") == " << std::boolalpha
        << (sub >= Mystr("aab")) << std::endl;

    std::cout << "\"aab\" >= sub == " << std::boolalpha
        << ("aab" >= sub) << std::endl;
    std::cout << "sub >= \"aab\" == " << std::boolalpha
        << (sub >= "aab") << std::endl;

    std::cout << "'a' >= sub == " << std::boolalpha
        << ('a' >= sub) << std::endl;
    std::cout << "sub >= 'a' == " << std::boolalpha
        << (sub >= 'a') << std::endl;

    return (0);
    }
```

Output

```
sub == aaa

sub >= sub == true
string("aab") >= sub == true
sub >= string("aab") == false
"aab" >= sub == true
sub >= "aab" == false
'a' >= sub == false
sub >= 'a' == true
```

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
regex_token_iterator Class
regex_traits Class
<regex> typedefs

`<regex>` typedefs

Article • 12/07/2021

cmatch
cregex_iterator
cregex_token_iterator
csub_match
regex
smatch
sregex_iterator
sregex_token_iterator
ssub_match
wcmatch
wcregex_iterator
wcregex_token_iterator
wcsub_match
wregex
wsmatch
wsregex_iterator
wsregex_token_iterator
wssub_match

cmatch Typedef

Type definition for char match_results.

```
C++
```

```
typedef match_results<const char*> cmatch;
```

Remarks

The type describes a specialization of class template match_results Class for iterators of type `const char*`.

cregex_iterator Typedef

Type definition for char regex_iterator.

```
C++
```

```
typedef regex_iterator<const char*> cregex_iterator;
```

Remarks

The type describes a specialization of class template regex_iterator Class for iterators of type `const char*`.

cregex_token_iterator Typedef

Type definition for char regex_token_iterator

```
C++
```

```
typedef regex_token_iterator<const char*> cregex_token_iterator;
```

Remarks

The type describes a specialization of class template regex_token_iterator Class for iterators of type `const char*`.

csub_match Typedef

Type definition for char sub_match.

```
C++
```

```
typedef sub_match<const char*> csub_match;
```

Remarks

The type describes a specialization of class template sub_match Class for iterators of type `const char*`.

regex Typedef

Type definition for char basic_regex.

```
```

```
C++
```

```
typedef basic_regex<char> regex;
```

Remarks

The type describes a specialization of class template basic_regex Class for elements of type `char`.

> **Note**
>
> High-bit characters will have unpredictable results with `regex`. Values outside the range of 0 to 127 may result in undefined behavior.

smatch Typedef

Type definition for string match_results.

```
C++
```

```
typedef match_results<string::const_iterator> smatch;
```

Remarks

The type describes a specialization of class template match_results Class for iterators of type `string::const_iterator`.

sregex_iterator Typedef

Type definition for string regex_iterator.

```
C++
```

```
typedef regex_iterator<string::const_iterator> sregex_iterator;
```

Remarks

The type describes a specialization of class template regex_iterator Class for iterators of type `string::const_iterator`.

sregex_token_iterator Typedef

Type definition for string regex_token_iterator.

```
C++

typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
```

Remarks

The type describes a specialization of class template regex_token_iterator Class for iterators of type `string::const_iterator`.

ssub_match Typedef

Type definition for string sub_match.

```
C++

typedef sub_match<string::const_iterator> ssub_match;
```

Remarks

The type describes a specialization of class template sub_match Class for iterators of type `string::const_iterator`.

wcmatch Typedef

Type definition for wchar_t match_results.

```
C++

typedef match_results<const wchar_t *> wcmatch;
```

Remarks

The type describes a specialization of class template match_results Class for iterators of type `const wchar_t*`.

wcregex_iterator Typedef

Type definition for wchar_t regex_iterator.

```
C++
```

```
typedef regex_iterator<const wchar_t*> wcregex_iterator;
```

Remarks

The type describes a specialization of class template regex_iterator Class for iterators of type `const wchar_t*`.

wcregex_token_iterator Typedef

Type definition for wchar_t regex_token_iterator.

```
C++
```

```
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
```

Remarks

The type describes a specialization of class template regex_token_iterator Class for iterators of type `const wchar_t*`.

wcsub_match Typedef

Type definition for wchar_t sub_match.

```
C++
```

```
typedef sub_match<const wchar_t*> wcsub_match;
```

Remarks

The type describes a specialization of class template sub_match Class for iterators of type `const wchar_t*`.

wregex Typedef

Type definition for wchar_t basic_regex.

```
C++

typedef basic_regex<wchar_t> wregex;
```

Remarks

The type describes a specialization of class template basic_regex Class for elements of type `wchar_t`.

wsmatch Typedef

Type definition for wstring match_results.

```
C++

typedef match_results<wstring::const_iterator> wsmatch;
```

Remarks

The type describes a specialization of class template match_results Class for iterators of type `wstring::const_iterator`.

wsregex_iterator Typedef

Type definition for wstring regex_iterator.

```
C++

typedef regex_iterator<wstring::const_iterator> wsregex_iterator;
```

Remarks

The type describes a specialization of class template regex_iterator Class for iterators of type `wstring::const_iterator`.

wsregex_token_iterator Typedef

Type definition for wstring regex_token_iterator.

```
C++
```

```
typedef regex_token_iterator<wstring::const_iterator>
wsregex_token_iterator;
```

Remarks

The type describes a specialization of class template regex_token_iterator Class for iterators of type `wstring::const_iterator`.

wssub_match Typedef

Type definition for wstring sub_match.

```
C++
```

```
typedef sub_match<wstring::const_iterator> wssub_match;
```

Remarks

The type describes a specialization of class template sub_match Class for iterators of type `wstring::const_iterator`.

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators

regex_token_iterator Class
regex_traits Class

basic_regex Class

Article • 06/16/2022

Wraps a regular expression.

Syntax

```
C++
```

```
template <class Elem, class RXtraits>
class basic_regex
```

Parameters

Elem
The type of elements to match.

RXtraits
Traits class for elements.

Remarks

The class template describes an object that holds a regular expression. Objects of this class template can be passed to the template functions regex_match, regex_search, and regex_replace. It also passes the suitable text string arguments, to search for text that matches the regular expression. There are two specializations of this class template, with the type definitions regex for elements of type `char`, and wregex for elements of type `wchar_t`.

The template argument *RXtraits* describes various important properties of the syntax of the regular expressions that the class template supports. A class that specifies these regular expression traits must have the same external interface as an object of type regex_traits Class.

Some functions take an operand sequence that defines a regular expression. You can specify such an operand sequence several ways:

`ptr`: a null-terminated sequence (such as a C string, for *Elem* of type `char`) beginning at `ptr` (which must not be a null pointer), where the terminating element is the value `value_type()` and isn't part of the operand sequence

`ptr`, `count`: a sequence of `count` elements beginning at `ptr` (which must not be a null pointer)

`str`: the sequence specified by the `basic_string` object `str`

`first`, `last`: a sequence of elements delimited by the iterators `first` and `last`, in the range `[first, last)`

`right`: the `basic_regex` object `right`

The above member functions also take an argument `flags` that specifies various options for the interpretation of the regular expression in addition to options described by the *RXtraits* type.

Members

Member	Default Value
public static const flag_type icase	regex_constants::icase
public static const flag_type nosubs	regex_constants::nosubs
public static const flag_type optimize	regex_constants::optimize
public static const flag_type collate	regex_constants::collate
public static const flag_type ECMAScript	regex_constants::ECMAScript
public static const flag_type basic	regex_constants::basic
public static const flag_type extended	regex_constants::extended
public static const flag_type awk	regex_constants::awk
public static const flag_type grep	regex_constants::grep
public static const flag_type egrep	regex_constants::egrep
private RXtraits traits	

Constructors

Constructor	Description
basic_regex	Construct the regular expression object.

Typedefs

Type name	Description
flag_type	The type of syntax option flags.
locale_type	The type of the stored locale object.
value_type	The element type.

Member functions

Member function	Description
assign	Assigns a value to the regular expression object.
flags	Returns syntax option flags.
getloc	Returns the stored locale object.
imbue	Alters the stored locale object.
mark_count	Returns number of subexpressions matched.
swap	Swaps two regular expression objects.

Operators

Operator	Description
operator=	Assigns a value to the regular expression object.

Requirements

Header: <regex>

Namespace: std

Example

C++

```
// std__regex__basic_regex.cpp
// compile with: /EHsc
#include <regex>
```

```cpp
#include <iostream>

using namespace std;

int main()
{
    regex::value_type elem = 'x';
    regex::flag_type flag = regex::grep;

    elem = elem;  // to quiet "unused" warnings
    flag = flag;

    // constructors
    regex rx0;
    cout << "match(\"abc\", \"\") == " << boolalpha
        << regex_match("abc", rx0) << endl;

    regex rx1("abcd", regex::ECMAScript);
    cout << "match(\"abc\", \"abcd\") == " << boolalpha
        << regex_match("abc", rx1) << endl;

    regex rx2("abcd", 3);
    cout << "match(\"abc\", \"abc\") == " << boolalpha
        << regex_match("abc", rx2) << endl;

    regex rx3(rx2);
    cout << "match(\"abc\", \"abc\") == " << boolalpha
        << regex_match("abc", rx3) << endl;

    string str("abcd");
    regex rx4(str);
    cout << "match(string(\"abcd\"), \"abc\") == " << boolalpha
        << regex_match("abc", rx4) << endl;

    regex rx5(str.begin(), str.end() - 1);
    cout << "match(string(\"abc\"), \"abc\") == " << boolalpha
        << regex_match("abc", rx5) << endl;
    cout << endl;

    // assignments
    rx0 = "abc";
    rx0 = rx1;
    rx0 = str;

    rx0.assign("abcd", regex::ECMAScript);
    rx0.assign("abcd", 3);
    rx0.assign(rx1);
    rx0.assign(str);
    rx0.assign(str.begin(), str.end() - 1);

    rx0.swap(rx1);

    // mark_count
    cout << "\"abc\" mark_count == "
        << regex("abc").mark_count() << endl;
```

```
    cout << "\"(abc)\" mark_count == "
        << regex("(abc)").mark_count() << endl;

    // locales
    regex::locale_type loc = rx0.imbue(locale());
    cout << "getloc == imbued == " << boolalpha
        << (loc == rx0.getloc()) << endl;

    // initializer_list
    regex rx6({ 'a', 'b', 'c' }, regex::ECMAScript);
    cout << "match(\"abc\") == " << boolalpha
        << regex_match("abc", rx6);
    cout << endl;
}
```

Output

```
match("abc", "") == false
match("abc", "abcd") == false
match("abc", "abc") == true
match("abc", "abc") == true
match(string("abcd"), "abc") == false
match(string("abc"), "abc") == true

"abc" mark_count == 0
"(abc)" mark_count == 1
getloc == imbued == true
match("abc") == true
```

basic_regex::assign

Assigns a value to the regular expression object.

C++

```
basic_regex& assign(
    const basic_regex& right);

basic_regex& assign(
    const Elem* ptr,
    flag_type flags = ECMAScript);

basic_regex& assign(
    const Elem* ptr,
    size_type len,
    flag_type flags = ECMAScript);

basic_regex& assign(
    initializer_list<_Elem> IList,
    flag_type flags = regex_constants::ECMAScript);
```

```
template <class STtraits, class STalloc>
basic_regex& assign(
    const basic_string<Elem, STtraits, STalloc>& str,
    flag_type flags = ECMAScript);

template <class InIt>
basic_regex& assign(
    InIt first, InIt last,
    flag_type flags = ECMAScript);
```

Parameters

STtraits
Traits class for a string source.

STalloc
Allocator class for a string source.

InIt
Input iterator type for a range source.

right
Regex source to copy.

ptr
Pointer to beginning of sequence to copy.

flags
Syntax option flags to add while copying.

len/TD>
Length of sequence to copy.

str
String to copy.

first
Beginning of sequence to copy.

last
End of sequence to copy.

IList
The initializer_list to copy.

Remarks

The member functions each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

basic_regex::basic_regex

Construct the regular expression object.

```
C++
```

```
basic_regex();

explicit basic_regex(
    const Elem* ptr,
    flag_type flags);

explicit basic_regex(
    const Elem* ptr,
    size_type len,
    flag_type flags);

basic_regex(
    const basic_regex& right);

basic_regex(
    initializer_list<Type> IList,
    flag_type flags);

template <class STtraits, class STalloc>
explicit basic_regex(
    const basic_string<Elem, STtraits, STalloc>& str,
    flag_type flags);

template <class InIt>
explicit basic_regex(
    InIt first,
    InIt last,
    flag_type flags);
```

Parameters

STtraits
Traits class for a string source.

STalloc
Allocator class for a string source.

InIt
Input iterator type for a range source.

right
Regex source to copy.

ptr
Pointer to beginning of sequence to copy.

flags
Syntax option flags to add while copying.

len/TD>
Length of sequence to copy.

str
String to copy.

first
Beginning of sequence to copy.

last
End of sequence to copy.

IList
The initializer_list to copy.

Remarks

All constructors store a default-constructed object of type `RXtraits`.

The first constructor constructs an empty `basic_regex` object. The other constructors construct a `basic_regex` object that holds the regular expression described by the operand sequence.

An empty `basic_regex` object doesn't match any character sequence when passed to regex_match, regex_search, or regex_replace.

basic_regex::flag_type

The type of syntax option flags.

```
C++
```

```
typedef regex_constants::syntax_option_type flag_type;
```

Remarks

The type is a synonym for regex_constants::syntax_option_type.

basic_regex::flags

Returns syntax option flags.

```
C++

flag_type flags() const;
```

Remarks

The member function returns the value of the `flag_type` argument passed to the most recent call to one of the basic_regex::assign member functions or, if no such call has been made, the value passed to the constructor.

basic_regex::getloc

Returns the stored locale object.

```
C++

locale_type getloc() const;
```

Remarks

The member function returns `traits.`regex_traits::getloc`()`.

basic_regex::imbue

Alters the stored locale object.

```
C++

locale_type imbue(locale_type loc);
```

Parameters

loc
The locale object to store.

Remarks

The member function empties `*this` and returns `traits.`regex_traits::imbue`(loc)`.

basic_regex::locale_type

The type of the stored locale object.

```
C++
```

```
typedef typename RXtraits::locale_type locale_type;
```

Remarks

The type is a synonym for regex_traits::locale_type.

basic_regex::mark_count

Returns number of subexpressions matched.

```
C++
```

```
unsigned mark_count() const;
```

Remarks

The member function returns the number of capture groups in the regular expression.

basic_regex::operator=

Assigns a value to the regular expression object.

```
C++
```

```
basic_regex& operator=(const basic_regex& right);

basic_regex& operator=(const Elem *str);

template <class STtraits, class STalloc>
basic_regex& operator=(const basic_string<Elem, STtraits, STalloc>& str);
```

Parameters

STtraits
Traits class for a string source.

STalloc
Allocator class for a string source.

right
Regex source to copy.

str
String to copy.

Remarks

The operators each replace the regular expression held by `*this` with the regular expression described by the operand sequence, then return `*this`.

basic_regex::swap

Swaps two regular expression objects.

C++

```
void swap(basic_regex& right) throw();
```

Parameters

right
The regular expression object to swap with.

Remarks

The member function swaps the regular expressions between `*this` and *right*. It does so in constant time and throws no exceptions.

basic_regex::value_type

The element type.

```
C++
```

```
typedef Elem value_type;
```

Remarks

The type is a synonym for the template parameter *Elem*.

See also

<regex>
regex_match
regex_search
regex_replace
regex
wregex
regex_traits Class

match_results Class

Article • 06/20/2022

Holds a sequence of submatches.

Syntax

```
C++
```

```
template <class BidIt, class Alloc>
class match_results
```

Parameters

BidIt
The iterator type for submatches.

Alloc
The type of an allocator for managing storage.

Remarks

The class template describes an object that controls a non-modifiable sequence of elements of type `sub_match<BidIt>` generated by a regular expression search. Each element points to the subsequence that matched the capture group corresponding to that element.

Constructors

Constructor	Description
match_results	Constructs the object.

Typedefs

Type name	Description
allocator_type	The type of an allocator for managing storage.

Type name	Description
char_type	The type of an element.
const_iterator	The const iterator type for submatches.
const_reference	The type of an element const reference.
difference_type	The type of an iterator difference.
iterator	The iterator type for submatches.
reference	The type of an element reference.
size_type	The type of a submatch count.
string_type	The type of a string.
value_type	The type of a submatch.

Member functions

Member function	Description
begin	Designates beginning of submatch sequence.
empty	Tests for no submatches.
end	Designates end of submatch sequence.
format	Formats submatches.
get_allocator	Returns the stored allocator.
length	Returns length of a submatch.
max_size	Gets largest number of submatches.
position	Get starting offset of a subgroup.
prefix	Gets sequence before first submatch.
size	Counts number of submatches.
str	Returns a submatch.
suffix	Gets sequence after last submatch.
swap	Swaps two match_results objects.

Operators

Operator	Description
operator=	Copy a match_results object.
operator[]	Access a subobject.

Requirements

Header: <regex>

Namespace: std

Example

C++

```
// std__regex__match_results.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
{
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;

    std::regex_search("xcaaay", mr, rx);

    std::cout << "prefix: matched == " << std::boolalpha
        << mr.prefix().matched
        << ", value == " << mr.prefix() << std::endl;
    std::cout << "whole match: " << mr.length() << " chars, value == "
        << mr.str() << std::endl;
    std::cout << "suffix: matched == " << std::boolalpha
        << mr.suffix().matched
        << ", value == " << mr.suffix() << std::endl;
    std::cout << std::endl;

    std::string fmt("\"c(a*)|(b)\" matched \"$&\"\n"
        "\"(a*)\" matched \"$1\"\n"
        "\"(b)\" matched \"$2\"\n");
    std::cout << mr.format(fmt) << std::endl;
    std::cout << std::endl;

    // index through submatches
    for (size_t n = 0; n < mr.size(); ++n)
    {
```

```
        std::cout << "submatch[" << n << "]: matched == " << std::boolalpha
            << mr[n].matched <<
            " at position " << mr.position(n) << std::endl;
        std::cout << "  " << mr.length(n)
            << " chars, value == " << mr[n] << std::endl;
    }
    std::cout << std::endl;

    // iterate through submatches
    for (std::cmatch::iterator it = mr.begin(); it != mr.end(); ++it)
    {
        std::cout << "next submatch: matched == " << std::boolalpha
            << it->matched << std::endl;
        std::cout << "  " << it->length()
            << " chars, value == " << *it << std::endl;
    }
    std::cout << std::endl;

    // other members
    std::cout << "empty == " << std::boolalpha << mr.empty() << std::endl;

    std::cmatch::allocator_type al = mr.get_allocator();
    std::cmatch::string_type str = std::string("x");
    std::cmatch::size_type maxsiz = mr.max_size();
    std::cmatch::char_type ch = 'x';
    std::cmatch::difference_type dif = mr.begin() - mr.end();
    std::cmatch::const_iterator cit = mr.begin();
    std::cmatch::value_type val = *cit;
    std::cmatch::const_reference cref = val;
    std::cmatch::reference ref = val;

    maxsiz = maxsiz;  // to quiet "unused" warnings
    if (ref == cref)
        ch = ch;
    dif = dif;

    return (0);
}
```

Output

```
prefix: matched == true, value == x
whole match: 4 chars, value == caaa
suffix: matched == true, value == y

"c(a*)|(b)" matched "caaa"
"(a*)" matched "aaa"
"(b)" matched ""

submatch[0]: matched == true at position 1
  4 chars, value == caaa
submatch[1]: matched == true at position 2
  3 chars, value == aaa
```

```
submatch[2]: matched == false at position 6
  0 chars, value ==

next submatch: matched == true
  4 chars, value == caaa
next submatch: matched == true
  3 chars, value == aaa
next submatch: matched == false
  0 chars, value ==

empty == false
```

match_results::allocator_type

The type of an allocator for managing storage.

C++

```
typedef Alloc allocator_type;
```

Remarks

The typedef is a synonym for the template argument *Alloc*.

match_results::begin

Designates beginning of submatch sequence.

C++

```
const_iterator begin() const;
```

Remarks

The member function returns a random access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

match_results::char_type

The type of an element.

C++

```
typedef typename iterator_traits<BidIt>::value_type char_type;
```

Remarks

The typedef is a synonym for the type `iterator_traits<BidIt>::value_type`, which is the element type of the character sequence that was searched.

match_results::const_iterator

The const iterator type for submatches.

```
C++
typedef T0 const_iterator;
```

Remarks

The typedef describes an object that can serve as a constant random-access iterator for the controlled sequence.

match_results::const_reference

The type of an element const reference.

```
C++
typedef const typename Alloc::const_reference const_reference;
```

Remarks

The typedef describes an object that can serve as a constant reference to an element of the controlled sequence.

match_results::difference_type

The type of an iterator difference.

```
C++
```

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

Remarks

The typedef is a synonym for the type `iterator_traits<BidIt>::difference_type`; it describes an object that can represent the difference between any two iterators that point at elements of the controlled sequence.

match_results::empty

Tests for no submatches.

```
C++
```

```
bool empty() const;
```

Remarks

The member function returns true only if the regular expression search failed.

match_results::end

Designates end of submatch sequence.

```
C++
```

```
const_iterator end() const;
```

Remarks

The member function returns an iterator that points just beyond the end of the sequence.

match_results::format

Formats submatches.

```
C++
```

```
template <class OutIt>
OutIt format(OutIt out,
    const string_type& fmt, match_flag_type flags = format_default) const;

string_type format(const string_type& fmt, match_flag_type flags =
format_default) const;
```

Parameters

OutIt
The output iterator type.

out
The output stream to write to.

fmt
The format string.

flags
The format flags.

Remarks

Each member function generates formatted text under the control of the format *fmt*. The first member function writes the formatted text to the sequence defined by its argument *out* and returns *out*. The second member function returns a string object holding a copy of the formatted text.

To generate formatted text. literal text in the format string is ordinarily copied to the target sequence. Each escape sequence in the format string is replaced by the text that it represents. The details of the copying and replacement are controlled by the format flags passed to the function.

match_results::get_allocator

Returns the stored allocator.

```
C++
```

```
allocator_type get_allocator() const;
```

Remarks

The member function returns a copy of the allocator object used by `*this` to allocate its `sub_match` objects.

match_results::iterator

The iterator type for submatches.

```
C++
```

```
typedef const_iterator iterator;
```

Remarks

The type describes an object that can serve as a random-access iterator for the controlled sequence.

match_results::length

Returns length of a submatch.

```
C++
```

```
difference_type length(size_type sub = 0) const;
```

Parameters

sub
The index of the submatch.

Remarks

The member function returns `(*this)[sub].length()`.

match_results::match_results

Constructs the object.

```
C++
```

```
explicit match_results(const Alloc& alloc = Alloc());

match_results(const match_results& right);
```

Parameters

alloc
The allocator object to store.

right
The match_results object to copy.

Remarks

The first constructor constructs a `match_results` object that holds no submatches. The second constructor constructs a `match_results` object that is a copy of *right*.

match_results::max_size

Gets largest number of submatches.

C++

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

match_results::operator=

Copy a match_results object.

C++

```
match_results& operator=(const match_results& right);
```

Parameters

right
The match_results object to copy.

Remarks

The member operator replaces the sequence controlled by `*this` with a copy of the sequence controlled by *right*.

match_results::operator[]

Access a subobject.

```
C++
```

```
const_reference operator[](size_type n) const;
```

Parameters

n
Index of the submatch.

Remarks

The member function returns a reference to element *n* of the controlled sequence, or a reference to an empty `sub_match` object if `size() <= n` or if capture group *n* wasn't part of the match.

match_results::position

Get starting offset of a subgroup.

```
C++
```

```
difference_type position(size_type sub = 0) const;
```

Parameters

sub
Index of the submatch.

Remarks

The member function returns `std::distance(prefix().first, (*this)[sub].first)`, that is, the distance from the first character in the target sequence to the first character in the submatch pointed to by element `n` of the controlled sequence.

match_results::prefix

Gets sequence before first submatch.

```
C++
```

```
const_reference prefix() const;
```

Remarks

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at the start of the target sequence and ends at `(*this)[0].first`, that is, it points to the text that precedes the matched subsequence.

match_results::reference

The type of an element reference.

```
C++
```

```
typedef const_reference reference;
```

Remarks

The type is a synonym for the type `const_reference`.

match_results::size

Counts number of submatches.

```
C++
```

```
size_type size() const;
```

Remarks

The member function returns one more than the number of capture groups in the regular expression that was used for the search, or zero if no search has been made.

match_results::size_type

The type of a submatch count.

```
C++
```

```
typedef typename Alloc::size_type size_type;
```

Remarks

The type is a synonym for the type `Alloc::size_type`.

match_results::str

Returns a submatch.

```
C++
```

```
string_type str(size_type sub = 0) const;
```

Parameters

sub
Index of the submatch.

Remarks

The member function returns `string_type((*this)[sub])`.

match_results::string_type

The type of a string.

```
C++
```

```
typedef basic_string<char_type> string_type;
```

Remarks

The type is a synonym for the type `basic_string<char_type>`.

match_results::suffix

Gets sequence after last submatch.

```
C++
```

```
const_reference suffix() const;
```

Remarks

The member function returns a reference to an object of type `sub_match<BidIt>` that points to the character sequence that begins at `(*this)[size() - 1].second` and ends at the end of the target sequence, that is, it points to the text that follows the matched subsequence.

match_results::swap

Swaps two match_results objects.

```
C++
```

```
void swap(const match_results& right) throw();
```

Parameters

right
The match_results object to swap with.

Remarks

The member function swaps the contents of `*this` and *right* in constant time and doesn't throw exceptions.

match_results::value_type

The type of a submatch.

```
C++
```

```
typedef sub_match<BidIt> value_type;
```

Remarks

The typedef is a synonym for the type `sub_match<BidIt>`.

See also

<regex>

regex_constants namespace

Article • 08/03/2021

Namespace for regular expression flags.

Syntax

```
C++
```

```
namespace regex_constants {
    enum syntax_option_type;
    enum match_flag_type;
    enum error_type;
}
```

Remarks

The namespace `regex_constants` encapsulates several flag types and their associated flag values.

Name	Description
error_type	Flags for reporting regular expression syntax errors.
match_flag_type	Flags for regular expression matching options.
syntax_option_type	Flags for selecting syntax options.

Requirements

Header: <regex>

Namespace: std

regex_constants::error_type

Flags for reporting regular expression syntax errors.

```
C++
```

```
enum error_type
    {    // identify error
    error_collate,
    error_ctype,
    error_escape,
    error_backref,
    error_brack,
    error_paren,
    error_brace,
    error_badbrace,
    error_range,
    error_space,
    error_badrepeat,
    error_complexity,
    error_stack,
    error_parse,
    error_syntax
    };
```

Remarks

The type is an enumerated type that describes an object that can hold error flags. The distinct flag values are:

`error_backref` -- the expression contained an invalid back reference

`error_badbrace` -- the expression contained an invalid count in a { } expression

`error_badrepeat` -- a repeat expression (one of '*', '', '+', '{' in most contexts) was not preceded by an expression

`error_brace` -- the expression contained an unmatched '{' or '}'

`error_brack` -- the expression contained an unmatched '[' or ']'

`error_collate` -- the expression contained an invalid collating element name

`error_complexity` -- an attempted match failed because it was too complex

`error_ctype` -- the expression contained an invalid character class name

`error_escape` -- the expression contained an invalid escape sequence

`error_paren` -- the expression contained an unmatched '(' or ')'

`error_parse` -- the expression failed to parse

`error_range` -- the expression contained an invalid character range specifier

`error_space` -- parsing a regular expression failed because there were not enough resources available

`error_stack` -- an attempted match failed because there was not enough memory available

`error_syntax` -- parsing failed on a syntax error

`error_backref` -- the expression contained an invalid back reference

regex_constants::match_flag_type

Flags for regular expression matching options.

C++

```
enum match_flag_type
    {    // specify matching and formatting rules
    match_default = 0x0000,
    match_not_bol = 0x0001,
    match_not_eol = 0x0002,
    match_not_bow = 0x0004,
    match_not_eow = 0x0008,
    match_any = 0x0010,
    match_not_null = 0x0020,
    match_continuous = 0x0040,
    match_prev_avail = 0x0100,
    format_default = 0x0000,
    format_sed = 0x0400,
    format_no_copy = 0x0800,
    format_first_only = 0x1000,
    _Match_not_null = 0x2000
    };
```

Remarks

The type is a bitmask type that describes options to be used when matching a text sequence against a regular expression and format flags to be used when replacing text. Options can be combined with `|`.

The match options are:

`match_default`

`match_not_bol` -- do not treat the first position in the target sequence as the beginning of a line

`match_not_eol` -- do not treat the past-the-end position in the target sequence as the end of a line

`match_not_bow` -- do not treat the first position in the target sequence as the beginning of a word

`match_not_eow` -- do not treat the past-the-end position in the target sequence as the end of a word

`match_any` -- if more than one match is possible any match is acceptable

`match_not_null` -- do not treat an empty subsequence as a match

`match_continuous` -- do not search for matches other than at the beginning of the target sequence

`match_prev_avail` -- `--first` is a valid iterator; ignore `match_not_bol` and `match_not_bow` if set

The format flags are:

`format_default` -- use ECMAScript format rules

`format_sed` -- use sed format rules

`format_no_copy` -- do not copy text that does not match the regular expression

`format_first_only` -- do not search for matches after the first one

regex_constants::syntax_option_type

Flags for selecting syntax options.

C++

```
enum syntax_option_type
    {    // specify RE syntax rules
    ECMAScript = 0x01,
    basic = 0x02,
    extended = 0x04,
    awk = 0x08,
    grep = 0x10,
    egrep = 0x20,
```

```
    _Gmask = 0x3F,

    icase = 0x0100,
    nosubs = 0x0200,
    optimize = 0x0400,
    collate = 0x0800
    };
```

Remarks

The type is a bitmask type that describes language specifiers and syntax modifiers to be used when compiling a regular expression. Options can be combined with `|`. No more than one language specifier should be used at a time.

The language specifiers are:

`ECMAScript` -- compile as ECMAScript

`basic` -- compile as BRE

`extended` -- compile as ERE

`awk` -- compile as awk

`grep` -- compile as grep

`egrep` -- compile as egrep

The syntax modifiers are:

`icase` -- make matches case-insensitive

`nosubs` -- the implementaton need not keep track of the contents of capture groups

`optimize` -- the implementation should emphasize speed of matching rather than speed of regular expression compilation

`collate` -- make matches locale-sensitive

See also

<regex>
regex_error Class
<regex> functions
regex_iterator Class

<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

regex_error Class

Article • 08/03/2021

Reports a bad basic_regex object.

Syntax

```
C++
```

```
class regex_error
: public std::runtime_error
```

Remarks

The class describes an exception object thrown to report an error in the construction or use of a `basic_regex` object.

Constructors

Constructor	Description
regex_error	Constructs the object.

Member functions

Member function	Description
code	Returns the error code.

Requirements

Header: <regex>

Namespace: std

Example

```
C++
```

```cpp
// std__regex__regex_error.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
    {
    std::regex_error paren(std::regex_constants::error_paren);

    try
        {
        std::regex rx("(a");
        }
    catch (const std::regex_error& rerr)
        {
        std::cout << "regex error: "
            << (rerr.code() == paren.code() ? "unbalanced parentheses" : "")
            << std::endl;
        }
    catch (...)
        {
        std::cout << "unknown exception" << std::endl;
        }

    return (0);
    }
```

Output

```
regex error: unbalanced parentheses
```

regex_error::code

Returns the error code.

C++

```cpp
regex_constants::error_code code() const;
```

Remarks

The member function returns the value that was passed to the object's constructor.

regex_error::regex_error

Constructs the object.

```
C++
```

```
regex_error(regex_constants::error_code error);
```

Parameters

error
The error code.

Remarks

The constructor constructs an object that holds the value *error*.

See also

<regex>
regex_constants Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

regex_iterator Class

Article • 06/20/2022

Iterator class for matches.

Syntax

```
C++
```

```
template<class BidIt,
   class Elem = typename std::iterator_traits<BidIt>::value_type,
   class RxTraits = regex_traits<Elem> >
class regex_iterator
```

Parameters

BidIt
The iterator type for submatches.

Elem
The type of elements to match.

RXtraits
Traits class for elements.

Remarks

The class template describes a constant forward iterator object. It extracts objects of type `match_results<BidIt>` by repeatedly applying its regular expression object `*pregex` to the character sequence defined by the iterator range `[begin, end)`.

Constructors

Constructor	Description
regex_iterator	Constructs the iterator.

Typedefs

Type name	Description
difference_type	The type of an iterator difference.
iterator_category	The type of the iterator category.
pointer	The type of a pointer to a match.
reference	The type of a reference to a match.
regex_type	The type of the regular expression to match.
value_type	The type of a match.

Operators

Operator	Description
operator!=	Compares iterators for inequality.
operator*	Accesses the designated match.
operator++	Increments the iterator.
operator=	Compares iterators for equality.
operator->	Accesses the designated match.

Requirements

Header: <regex>

Namespace: std

Examples

See the following articles for examples on regular expressions:

- regex_match
- regex_replace
- regex_search
- swap

C++

```
// std__regex__regex_iterator.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::regex_iterator<const char *> Myiter;
int main()
    {
    const char *pat = "axayaz";
    Myiter::regex_type rx("a");
    Myiter next(pat, pat + strlen(pat), rx);
    Myiter end;

    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;

// other members
    Myiter it1(pat, pat + strlen(pat), rx);
    Myiter it2(it1);
    next = it1;

    Myiter::iterator_category cat = std::forward_iterator_tag();
    Myiter::difference_type dif = -3;
    Myiter::value_type mr = *it1;
    Myiter::reference ref = mr;
    Myiter::pointer ptr = &ref;

    dif = dif; // to quiet "unused" warnings
    ptr = ptr;

    return (0);
    }
```

Output

```
match == a
match == a
match == a
```

regex_iterator::difference_type

The type of an iterator difference.

C++

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The type is a synonym for `std::ptrdiff_t`.

regex_iterator::iterator_category

The type of the iterator category.

```
C++
```

```
typedef std::forward_iterator_tag iterator_category;
```

Remarks

The type is a synonym for `std::forward_iterator_tag`.

regex_iterator::operator!=

Compares iterators for inequality.

```
C++
```

```
bool operator!=(const regex_iterator& right);
```

Parameters

right
The iterator to compare to.

Remarks

The member function returns `!(*this == right)`.

regex_iterator::operator*

Accesses the designated match.

```
C++
```

```
const match_results<BidIt>& operator*();
```

Remarks

The member function returns the stored value `match`.

regex_iterator::operator++

Increments the iterator.

```
C++
```

```
regex_iterator& operator++();
regex_iterator& operator++(int);
```

Remarks

If the current match has no characters, the first operator calls `regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail | regex_constants::match_not_null)`; otherwise it advances the stored value `begin` to point to the first character after the current match then calls `regex_search(begin, end, match, *pregex, flags | regex_constants::match_prev_avail)`. In either case, if the search fails the operator sets the object to an end-of-sequence iterator. The operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_iterator::operator=

Compares iterators for equality.

```
C++
```

```
bool operator==(const regex_iterator& right);
```

Parameters

right
The iterator to compare to.

Remarks

The member function returns true if `*this` and *right* are both end-of-sequence iterators or if neither is an end-of-sequence iterator and `begin == right.begin`, `end == right.end`, `pregex == right.pregex`, and `flags == right.flags`. Otherwise it returns false.

`regex_iterator::operator->`

Accesses the designated match.

```
C++
```

```
const match_results<BidIt> * operator->();
```

Remarks

The member function returns the address of the stored value `match`.

regex_iterator::pointer

The type of a pointer to a match.

```
C++
```

```
typedef match_results<BidIt> *pointer;
```

Remarks

The type is a synonym for `match_results<BidIt>*`, where `BidIt` is the template parameter.

regex_iterator::reference

The type of a reference to a match.

```
C++
```

```
typedef match_results<BidIt>& reference;
```

Remarks

The type is a synonym for `match_results<BidIt>&`, where `BidIt` is the template parameter.

regex_iterator::regex_iterator

Constructs the iterator.

```
C++
```

```
regex_iterator();

regex_iterator(BidIt first,
    BidIt last,
    const regex_type& re,
    regex_constants::match_flag_type f = regex_constants::match_default);
```

Parameters

first
Beginning of sequence to match.

last
End of sequence to match.

re
Regular expression for matches.

f
Flags for matches.

Remarks

The first constructor constructs an end-of-sequence iterator. The second constructor initializes the stored value `begin` with *first*, the stored value `end` with *last*, the stored value `pregex` with `&re`, and the stored value `flags` with *f*. It then calls `regex_search(begin, end, match, *pregex, flags)`. If the search fails, the constructor sets the object to an end-of-sequence iterator.

regex_iterator::regex_type

The type of the regular expression to match.

```
C++
```

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

Remarks

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_iterator::value_type

The type of a match.

```
C++
```

```
typedef match_results<BidIt> value_type;
```

Remarks

The type is a synonym for `match_results<BidIt>`, where `BidIt` is the template parameter.

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

regex_token_iterator Class

Article • 12/07/2021

Iterator class for submatches.

Syntax

```
C++
```

```
template<class BidIt,
   class Elem = typename std::iterator_traits<BidIt>::value_type,
   class RxTraits = regex_traits<Elem> >
class regex_token_iterator
```

Parameters

BidIt
The iterator type for submatches.

Elem
The type of elements to match.

RXtraits
Traits class for elements.

Remarks

The class template describes a constant forward iterator object. Conceptually, it holds a `regex_iterator` object that it uses to search for regular expression matches in a character sequence. It extracts objects of type `sub_match<BidIt>` representing the submatches identified by the index values in the stored vector `subs` for each regular expression match.

An index value of -1 designates the character sequence beginning immediately after the end of the previous regular expression match, or beginning at the start of the character sequence if there was no previous regular expression match, and extending to but not including the first character of the current regular expression match, or to the end of the character sequence if there is no current match. Any other index value `idx` designates the contents of the capture group held in `it.match[idx]`.

Members

Member	Default Value
`private regex_iterator<BidIt, Elem, RXtraits> it`	
`private vector<int> subs`	
`private int pos`	

Constructors

Constructor	Description
regex_token_iterator	Constructs the iterator.

Typedefs

Type name	Description
difference_type	The type of an iterator difference.
iterator_category	The type of the iterator category.
pointer	The type of a pointer to a match.
reference	The type of a reference to a submatch.
regex_type	The type of the regular expression to match.
value_type	The type of a submatch.

Operators

Operator	Description
operator!=	Compares iterators for inequality.
operator*	Accesses the designated submatch.
operator++	Increments the iterator.
operator==	Compares iterators for equality.
operator->	Accesses the designated submatch.

Requirements

Header: <regex>

Namespace: std

Example

C++

```
#include <regex>
#include <iostream>

typedef std::regex_token_iterator<const char *> Myiter;
int main()
    {
    const char *pat = "aaxaayaaz";
    Myiter::regex_type rx("(a)a");
    Myiter next(pat, pat + strlen(pat), rx);
    Myiter end;

// show whole match
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

// show prefix before match
    next = Myiter(pat, pat + strlen(pat), rx, -1);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

// show (a) submatch only
    next = Myiter(pat, pat + strlen(pat), rx, 1);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

// show prefixes and submatches
    std::vector<int> vec;
    vec.push_back(-1);
    vec.push_back(1);
    next = Myiter(pat, pat + strlen(pat), rx, vec);
    for (; next != end; ++next)
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

// show prefixes and whole matches
    int arr[] = {-1, 0};
    next = Myiter(pat, pat + strlen(pat), rx, arr);
    for (; next != end; ++next)
```

```
        std::cout << "match == " << next->str() << std::endl;
    std::cout << std::endl;

// other members
    Myiter it1(pat, pat + strlen(pat), rx);
    Myiter it2(it1);
    next = it1;

    Myiter::iterator_category cat = std::forward_iterator_tag();
    Myiter::difference_type dif = -3;
    Myiter::value_type mr = *it1;
    Myiter::reference ref = mr;
    Myiter::pointer ptr = &ref;

    dif = dif; // to quiet "unused" warnings
    ptr = ptr;

    return (0);
    }
```

Output

```
match == aa
match == aa
match == aa

match ==
match == x
match == y
match == z

match == a
match == a
match == a

match ==
match == a
match == x
match == a
match == y
match == a
match == z

match ==
match == aa
match == x
match == aa
match == y
match == aa
match == z
```

regex_token_iterator::difference_type

The type of an iterator difference.

```
C++
```

```
typedef std::ptrdiff_t difference_type;
```

Remarks

The type is a synonym for `std::ptrdiff_t`.

regex_token_iterator::iterator_category

The type of the iterator category.

```
C++
```

```
typedef std::forward_iterator_tag iterator_category;
```

Remarks

The type is a synonym for `std::forward_iterator_tag`.

regex_token_iterator::operator!=

Compares iterators for inequality.

```
C++
```

```
bool operator!=(const regex_token_iterator& right);
```

Parameters

right
The iterator to compare to.

Remarks

The member function returns `!(*this == right)`.

regex_token_iterator::operator*

Accesses the designated submatch.

```
C++
```

```
const sub_match<BidIt>& operator*();
```

Remarks

The member function returns a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

regex_token_iterator::operator++

Increments the iterator.

```
C++
```

```
regex_token_iterator& operator++();

regex_token_iterator& operator++(int);
```

Remarks

If the stored iterator `it` is an end-of-sequence iterator the first operator sets the stored value `pos` to the value of `subs.size()` (thus making an end-of-sequence iterator). Otherwise the operator increments the stored value `pos`; if the result is equal to the value `subs.size()` it sets the stored value `pos` to zero and increments the stored iterator `it`. If incrementing the stored iterator leaves it unequal to an end-of-sequence iterator the operator does nothing further. Otherwise, if the end of the preceding match was at the end of the character sequence the operator sets the stored value of `pos` to `subs.size()`. Otherwise, the operator repeatedly increments the stored value `pos` until `pos == subs.size()` or `subs[pos] == -1` (thus ensuring that the next dereference of the iterator will return the tail of the character sequence if one of the index values is -1). In all cases the operator returns the object.

The second operator makes a copy of the object, increments the object, then returns the copy.

regex_token_iterator::operator==

Compares iterators for equality.

```
C++
```

```
bool operator==(const regex_token_iterator& right);
```

Parameters

right
The iterator to compare to.

Remarks

The member function returns `it == right.it && subs == right.subs && pos == right.pos`.

`regex_token_iterator::operator->`

Accesses the designated submatch.

```
C++
```

```
const sub_match<BidIt> * operator->();
```

Remarks

The member function returns a pointer to a `sub_match<BidIt>` object representing the capture group identified by the index value `subs[pos]`.

regex_token_iterator::pointer

The type of a pointer to a match.

```
C++
```

```
typedef sub_match<BidIt> *pointer;
```

Remarks

The type is a synonym for `sub_match<BidIt>*`, where `BidIt` is the template parameter.

regex_token_iterator::reference

The type of a reference to a submatch.

C++

```
typedef sub_match<BidIt>& reference;
```

Remarks

The type is a synonym for `sub_match<BidIt>&`, where `BidIt` is the template parameter.

regex_token_iterator::regex_token_iterator

Constructs the iterator.

C++

```
regex_token_iterator();

regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, int submatch = 0,
    regex_constants::match_flag_type f = regex_constants::match_default);

regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const vector<int> submatches,
    regex_constants::match_flag_type f = regex_constants::match_default);

template <std::size_t N>
regex_token_iterator(BidIt first, BidIt last,
    const regex_type& re, const int (&submatches)[N],
    regex_constants::match_flag_type f = regex_constants::match_default);
```

Parameters

first
Beginning of sequence to match.

last
End of sequence to match.

re
Regular expression for matches.

f
Flags for matches.

Remarks

The first constructor constructs an end-of-sequence iterator.

The second constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds exactly one integer, with value `submatch`, and whose stored value `pos` is zero. Note: the resulting object extracts the submatch identified by the index value `submatch` for each successful regular expression match.

The third constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds a copy of the constructor argument `submatches`, and whose stored value `pos` is zero.

The fourth constructor constructs an object whose stored iterator `it` is initialized to `regex_iterator<BidIt, Elem, RXtraits>(first, last, re, f)`, whose stored vector `subs` holds the `N` values pointed to by the constructor argument `submatches`, and whose stored value `pos` is zero.

regex_token_iterator::regex_type

The type of the regular expression to match.

```
C++
```

```
typedef basic_regex<Elem, RXtraits> regex_type;
```

Remarks

The typedef is a synonym for `basic_regex<Elem, RXtraits>`.

regex_token_iterator::value_type

The type of a submatch.

```
C++
```

```
typedef sub_match<BidIt> value_type;
```

Remarks

The type is a synonym for `sub_match<BidIt>`, where `BidIt` is the template parameter.

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_traits Class
<regex> typedefs

regex_traits Class

Article • 08/03/2021

Describes characteristics of elements for matching.

Syntax

```
C++
```

```
template<class Elem>
class regex_traits
```

Parameters

Elem
The character element type to describe.

Remarks

The class template describes various regular expression traits for type *Elem*. The class template basic_regex Class uses this information to manipulate elements of type *Elem*.

Each `regex_traits` object holds an object of type `regex_traits::locale` which is used by some of its member functions. The default locale is a copy of `regex_traits::locale()`. The member function `imbue` replaces the locale object, and the member function `getloc` returns a copy of the locale object.

Constructors

Constructor	Description
regex_traits	Constructs the object.

Typedefs

Type name	Description
char_class_type	The type of character class designators.

Type name	Description
char_type	The type of an element.
locale_type	The type of the stored locale object.
size_type	The type of a sequence length.
string_type	The type of a string of elements.

Member functions

Member function	Description
getloc	Returns the stored locale object.
imbue	Alters the stored locale object.
isctype	Tests for class membership.
length	Returns the length of a null-terminated sequence.
lookup_classname	Maps a sequence to a character class.
lookup_collatename	Maps a sequence to a collating element.
transform	Converts to equivalent ordered sequence.
transform_primary	Converts to equivalent caseless ordered sequence.
translate	Converts to equivalent matching element.
translate_nocase	Converts to equivalent caseless matching element.
value	Converts an element to a digit value.

Requirements

Header: <regex>

Namespace: std

Example

C++

```cpp
// std__regex__regex_traits.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

typedef std::regex_traits<char> Mytr;
int main()
    {
    Mytr tr;

    Mytr::char_type ch = tr.translate('a');
    std::cout << "translate('a') == 'a' == " << std::boolalpha
        << (ch == 'a') << std::endl;

    std::cout << "nocase 'a' == 'A' == " << std::boolalpha
        << (tr.translate_nocase('a') == tr.translate_nocase('A'))
        << std::endl;

    const char *lbegin = "abc";
    const char *lend = lbegin + strlen(lbegin);
    Mytr::size_type size = tr.length(lbegin);
    std::cout << "length(\"abc\") == " << size <<std::endl;

    Mytr::string_type str = tr.transform(lbegin, lend);
    std::cout << "transform(\"abc\") < \"abc\" == " << std::boolalpha
        << (str < "abc") << std::endl;

    const char *ubegin = "ABC";
    const char *uend = ubegin + strlen(ubegin);
    std::cout << "primary \"ABC\" < \"abc\" == " << std::boolalpha
        << (tr.transform_primary(ubegin, uend) <
            tr.transform_primary(lbegin, lend))
        << std::endl;

    const char *dig = "digit";
    Mytr::char_class_type cl = tr.lookup_classname(dig, dig + 5);
    std::cout << "class digit == d == " << std::boolalpha
        << (cl == tr.lookup_classname(dig, dig + 1))
        << std::endl;

    std::cout << "'3' is digit == " <<std::boolalpha
        << tr.isctype('3', tr.lookup_classname(dig, dig + 5))
        << std::endl;

    std::cout << "hex C == " << tr.value('C', 16) << std::endl;

// other members
    str = tr.lookup_collatename(dig, dig + 5);

    Mytr::locale_type loc = tr.getloc();
    tr.imbue(loc);

    return (0);
    }
```

Output

```
translate('a') == 'a' == true
nocase 'a' == 'A' == true
length("abc") == 3
transform("abc") < "abc" == false
primary "ABC" < "abc" == false
class digit == d == true
'3' is digit == true
hex C == 12
```

regex_traits::char_class_type

The type of character class designators.

C++

```
typedef T8 char_class_type;
```

Remarks

The type is a synonym for an unspecified type that designates character classes. Values of this type can be combined using the `|` operator to designate character classes that are the union of the classes designated by the operands.

regex_traits::char_type

The type of an element.

C++

```
typedef Elem char_type;
```

Remarks

The typedef is a synonym for the template argument `Elem`.

regex_traits::getloc

Returns the stored locale object.

```
C++
```

```
locale_type getloc() const;
```

Remarks

The member function returns the stored `locale` object.

regex_traits::imbue

Alters the stored locale object.

```
C++
```

```
locale_type imbue(locale_type loc);
```

Parameters

loc
The locale object to store.

Remarks

The member function copies *loc* to the stored `locale` object and returns a copy of the previous value of the stored `locale` object.

regex_traits::isctype

Tests for class membership.

```
C++
```

```
bool isctype(char_type ch, char_class_type cls) const;
```

Parameters

ch
The element to test.

cls
The classes to test for.

Remarks

The member function returns true only if the character *ch* is in the character class designated by *cls*.

regex_traits::length

Returns the length of a null-terminated sequence.

```
C++
```

```
static size_type length(const char_type *str);
```

Parameters

str
The null-terminated sequence.

Remarks

The static member function returns `std::char_traits<char_type>::length(str)`.

regex_traits::locale_type

The type of the stored locale object.

```
C++
```

```
typedef T7 locale_type;
```

Remarks

The typedef is a synonym for a type that encapsulates locales. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::locale`.

regex_traits::lookup_classname

Maps a sequence to a character class.

```
C++
```

```
template <class FwdIt>
char_class_type lookup_classname(FwdIt first, FwdIt last) const;
```

Parameters

first
Beginning of sequence to look up.

last
End of sequence to look up.

Remarks

The member function returns a value that designates the character class named by the character sequence pointed to by its arguments. The value does not depend on the case of the characters in the sequence.

The specialization `regex_traits<char>` recognizes the names `"d"`, `"s"`, `"w"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"`, all without regard to case.

The specialization `regex_traits<wchar_t>` recognizes the names `L"d"`, `L"s"`, `L"w"`, `L"alnum"`, `L"alpha"`, `L"blank"`, `L"cntrl"`, `L"digit"`, `L"graph"`, `L"lower"`, `L"print"`, `L"punct"`, `L"space"`, `L"upper"`, and `L"xdigit"`, all without regard to case.

regex_traits::lookup_collatename

Maps a sequence to a collating element.

```
C++
```

```
template <class FwdIt>
string_type lookup_collatename(FwdIt first, FwdIt last) const;
```

Parameters

first
Beginning of sequence to look up.

last
End of sequence to look up.

Remarks

The member function returns a string object containing the collating element corresponding to the sequence `[first, last)`, or an empty string if the sequence is not a valid collating element.

regex_traits::regex_traits

Constructs the object.

```
C++
```

```
regex_traits();
```

Remarks

The constructor constructs an object whose stored `locale` object is initialized to the default locale.

regex_traits::size_type

The type of a sequence length.

```
C++
```

```
typedef T6 size_type;
```

Remarks

The typedef is a synonym for an unsigned integral type. In the specializations `regex_traits<char>` and `regex_traits<wchar_t>` it is a synonym for `std::size_t`.

The typedef is a synonym for `std::size_t`.

regex_traits::string_type

The type of a string of elements.

```
C++
```

```
typedef basic_string<Elem> string_type;
```

Remarks

The typedef is a synonym for `basic_string<Elem>`.

regex_traits::transform

Converts to equivalent ordered sequence.

```
C++
```

```
template <class FwdIt>
string_type transform(FwdIt first, FwdIt last) const;
```

Parameters

first
Beginning of sequence to transform.

last
End of sequence to transform.

Remarks

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform(first1, last1) < transform(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)`.

regex_traits::transform_primary

Converts to equivalent caseless ordered sequence.

```
C++
```

```
template <class FwdIt>
string_type transform_primary(FwdIt first, FwdIt last) const;
```

Parameters

first
Beginning of sequence to transform.

last
End of sequence to transform.

Remarks

The member function returns a string that it generates by using a transformation rule that depends on the stored `locale` object. For two character sequences designated by the iterator ranges `[first1, last1)` and `[first2, last2)`, `transform_primary(first1, last1) < transform_primary(first2, last2)` if the character sequence designated by the iterator range `[first1, last1)` sorts before the character sequence designated by the iterator range `[first2, last2)` without regard for case or accents.

regex_traits::translate

Converts to equivalent matching element.

```
C++
```

```
char_type translate(char_type ch) const;
```

Parameters

ch
The element to convert.

Remarks

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate(ch1) == translate(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a locale-sensitive match.

regex_traits::translate_nocase

Converts to equivalent caseless matching element.

```
C++
```

```
char_type translate_nocase(char_type ch) const;
```

Parameters

ch
The element to convert.

Remarks

The member function returns a character that it generates by using a transformation rule that depends on the stored `locale` object. For two `char_type` objects `ch1` and `ch2`, `translate_nocase(ch1) == translate_nocase(ch2)` only if `ch1` and `ch2` should match when one occurs in the regular expression definition and the other occurs at a corresponding position in the target sequence for a case-insensitive match.

regex_traits::value

Converts an element to a digit value.

```
C++
```

```
int value(Elem ch, int radix) const;
```

Parameters

ch
The element to convert.

radix
The arithmetic base to use.

Remarks

The member function returns the value represented by the character *ch* in the base *radix*, or -1 if *ch* is not a valid digit in the base *radix*. The function will only be called with a *radix* argument of 8, 10, or 16.

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
<regex> typedefs
regex_traits<char> Class
regex_traits<wchar_t> Class

`regex_traits<char>` Class

Article • 12/06/2021

Specialization of `regex_traits` for `char`.

Syntax

```
C++
```

```
template <>
class regex_traits<char>
```

Remarks

The class is an explicit specialization of class template regex_traits for elements of type `char` (so that it can take advantage of library functions that manipulate objects of this type).

Requirements

Header: <regex>

Namespace: std

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

`regex_traits<wchar_t>` Class

Article • 12/06/2021

Specialization of `regex_traits` for `wchar_t`.

Syntax

```
C++
```

```
template <>
class regex_traits<wchar_t>
```

Remarks

The class is an explicit specialization of class template regex_traits for elements of type `wchar_t` (so that it can take advantage of library functions that manipulate objects of this type).

Requirements

Header: <regex>

Namespace: std

See also

<regex>
regex_constants Class
regex_error Class
<regex> functions
regex_iterator Class
<regex> operators
regex_token_iterator Class
regex_traits Class
<regex> typedefs

sub_match Class

Article • 06/21/2022

Describes a submatch.

Syntax

```
C++
```

```
template <class BidIt>
class sub_match
    : public std::pair<BidIt, BidIt>
```

Parameters

BidIt
The iterator type for submatches.

Remarks

The class template describes an object that designates a sequence of characters that matched a capture group in a call to regex_match or to regex_search. Objects of type match_results Class hold an array of these objects, one for each capture group in the regular expression that was used in the search.

If the capture group wasn't matched the object's data member `matched` holds false, and the two iterators `first` and `second` (inherited from the base `std::pair`) are equal. If the capture group was matched, `matched` holds true, the iterator `first` points to the first character in the target sequence that matched the capture group, and the iterator `second` points one position past the last character in the target sequence that matched the capture group. For a zero-length match, the member `matched` holds true, the two iterators will be equal, and both will point to the position of the match.

A zero-length match can occur when a capture group consists solely of an assertion, or of a repetition that allows zero repeats. For example:

"^" matches the target sequence "a"; the `sub_match` object corresponding to capture group 0 holds iterators that both point to the first character in the sequence.

"b(a*)b" matches the target sequence "bb"; the `sub_match` object corresponding to capture group 1 holds iterators that both point to the second character in the sequence.

Typedefs

Type name	Description
difference_type	The type of an iterator difference.
iterator	The type of an iterator.
value_type	The type of an element.

Member functions

Member function	Description
compare	Compare submatch against a sequence.
length	Returns the length of a submatch.
matched	Indicates if match succeeded.
str	Converts submatch to a string.

Operators

Operator	Description
operator basic_string<value_type>	Casts submatch to a string.

Example

C++

```
// std__regex__sub_match.cpp
// compile with: /EHsc
#include <regex>
#include <iostream>

int main()
    {
    std::regex rx("c(a*)|(b)");
    std::cmatch mr;
```

```
    std::regex_search("xcaaay", mr, rx);

    std::csub_match sub = mr[1];
    std::cout << "matched == " << std::boolalpha
        << sub.matched << std::endl;
    std::cout << "length == " << sub.length() << std::endl;

    std::csub_match::difference_type dif = std::distance(sub.first,
sub.second);
    std::cout << "difference == " << dif << std::endl;

    std::csub_match::iterator first = sub.first;
    std::csub_match::iterator last = sub.second;
    std::cout << "range == " << std::string(first, last)
        << std::endl;
    std::cout << "string == " << sub << std::endl;

    std::csub_match::value_type const *ptr = "aab";
    std::cout << "compare(\"aab\") == "
        << sub.compare(ptr) << std::endl;
    std::cout << "compare(string) == "
        << sub.compare(std::string("AAA")) << std::endl;
    std::cout << "compare(sub) == "
        << sub.compare(sub) << std::endl;

    return (0);
    }
```

Output

```
matched == true
length == 3
difference == 3
range == aaa
string == aaa
compare("aab") == -1
compare(string) == 1
compare(sub) == 0
```

Requirements

Header: <regex>

Namespace: std

sub_match::compare

Compare submatch against a sequence.

C++

```
int compare(const sub_match& right) const;
int compare(const basic_string<value_type>& str) const;
int compare(const value_type *ptr) const;
```

Parameters

right
The submatch to compare to.

str
The string to compare to.

ptr
The null-terminated sequence to compare to.

Remarks

The first member function compares the matched sequence `[first, second)` to the matched sequence `[right.first, right.second)`. The second member function compares the matched sequence `[first, second)` to the character sequence `[right.begin(), right.end())`. The third member function compares the matched sequence `[first, second)` to the character sequence `[right, right + std::char_traits<value_type>::length(right))`.

Each function returns:

a negative value if the first differing value in the matched sequence compares less than the corresponding element in the operand sequence (as determined by `std::char_traits<value_type>::compare`), or if the two have a common prefix but the target sequence is longer

zero if the two compare equal element by element and have the same length

a positive value otherwise

sub_match::difference_type

The type of an iterator difference.

C++

```
typedef typename iterator_traits<BidIt>::difference_type difference_type;
```

Remarks

The typedef is a synonym for `iterator_traits<BidIt>::difference_type`.

sub_match::iterator

The type of an iterator.

```
C++
typedef BidIt iterator;
```

Remarks

The typedef is a synonym for the template type argument `Bidit`.

sub_match::length

Returns the length of a submatch.

```
C++
difference_type length() const;
```

Remarks

The member function returns the length of the matched sequence, or zero if there was no matched sequence.

sub_match::matched

Indicates if match succeeded.

```
C++
bool matched;
```

Remarks

The member holds `true` only if the capture group associated with `*this` was part of the regular expression match.

`sub_match::operator basic_string<value_type>`

Casts submatch to a string.

```
C++
```

```
operator basic_string<value_type>() const;
```

Remarks

The member operator returns `str()`.

sub_match::str

Converts submatch to a string.

```
C++
```

```
basic_string<value_type> str() const;
```

Remarks

The member function returns `basic_string<value_type>(first, second)`.

sub_match::value_type

The type of an element.

```
C++
```

```
typedef typename iterator_traits<BidIt>::value_type value_type;
```

Remarks

The typedef is a synonym for `iterator_traits<BidIt>::value_type`.

See also

<regex>
sub_match

<scoped_allocator>

Article • 12/07/2021

Defines the container class template scoped_allocator.

Syntax

```
C++

#include <scoped_allocator>
```

Operators

Operator	Description
operator!=	Tests if the scoped_allocator object on the left side of the operator is not equal to the list object on the right side.
operator==	Tests if the scoped_allocator object on the left side of the operator is equal to the list object on the right side.

Classes

Class	Description
scoped_allocator_adaptor Class	A class template that encapsulates a nest of one or more allocators.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<scoped_allocator>` operators

Article • 12/07/2021

operator!=
operator==

operator!=

Tests two `scoped_allocator_adaptor` objects for inequality.

```
C++
```

```
template <class Outer, class... Inner>
bool operator!=(
    const scoped_allocator_adaptor<Outer, Inner...>& left,
    const scoped_allocator_adaptor<Outer, Inner...>& right) noexcept;
```

Parameters

left
The left `scoped_allocator_adaptor` object.

right
The right `scoped_allocator_adaptor` object.

Return Value

`!(left == right)`

operator==

Tests two `scoped_allocator_adaptor` objects for equality.

```
C++
```

```
template <class Outer, class... Inner>
bool operator==(
    const scoped_allocator_adaptor<Outer, Inner...>& left,
    const scoped_allocator_adaptor<Outer, Inner...>& right) noexcept;
```

Parameters

left
The left `scoped_allocator_adaptor` object.

right
The right `scoped_allocator_adaptor` object.

Return Value

`left.outer_allocator() == right.outer_allocator() && left.inner_allocator() == right.inner_allocator()`

See also

<scoped_allocator>

scoped_allocator_adaptor Class

Article • 06/21/2022

Represents a nest of allocators.

Syntax

```
C++
```

```
template <class Outer, class... Inner>
class scoped_allocator_adaptor;
```

Remarks

The class template encapsulates a nest of one or more allocators. Each such class has an outermost allocator of type `outer_allocator_type`, a synonym for `Outer`, which is a public base of the `scoped_allocator_adaptor` object. `Outer` is used to allocate memory to be used by a container. You can obtain a reference to this allocator base object by calling `outer_allocator`.

The remainder of the nest has type `inner_allocator_type`. An inner allocator is used to allocate memory for elements within a container. You can obtain a reference to the stored object of this type by calling `inner_allocator`. If `Inner...` isn't empty, `inner_allocator_type` has type `scoped_allocator_adaptor<Inner...>`, and `inner_allocator` designates a member object. Otherwise, `inner_allocator_type` has type `scoped_allocator_adaptor<Outer>`, and `inner_allocator` designates the entire object.

The nest behaves as if it has arbitrary depth, replicating its innermost encapsulated allocator as needed.

Several concepts that aren't a part of the visible interface aid in describing the behavior of this class template. An *outermost allocator* mediates all calls to the construct and destroy methods. It's effectively defined by the recursive function `OUTERMOST(X)`, where `OUTERMOST(X)` is one of the following.

- If `X.outer_allocator()` is well formed, then `OUTERMOST(X)` is `OUTERMOST(X.outer_allocator())`.
- Otherwise, `OUTERMOST(X)` is `X`.

Three types are defined for the sake of exposition:

Type	Description
`Outermost`	The type of `OUTERMOST(*this)`.
`Outermost_traits`	`allocator_traits<Outermost>`
`Outer_traits`	`allocator_traits<Outer>`

Constructors

Name	Description
scoped_allocator_adaptor	Constructs a `scoped_allocator_adaptor` object.

Typedefs

Name	Description
`const_pointer`	This type is a synonym for the `const_pointer` that is associated with the allocator `Outer`.
`const_void_pointer`	This type is a synonym for the `const_void_pointer` that is associated with the allocator `Outer`.
`difference_type`	This type is a synonym for the `difference_type` that is associated with the allocator `Outer`.
`inner_allocator_type`	This type is a synonym for the type of the nested adaptor `scoped_allocator_adaptor<Inner...>`.
`outer_allocator_type`	This type is a synonym for the type of the base allocator `Outer`.
`pointer`	This type is a synonym for the `pointer` associated with the allocator `Outer`.
`propagate_on_container_copy_assignment`	The type holds true only if `Outer_traits::propagate_on_container_copy_assignment` holds true or `inner_allocator_type::propagate_on_container_copy_assignment` holds true.
`propagate_on_container_move_assignment`	The type holds true only if `Outer_traits::propagate_on_container_move_assignment` holds true or `inner_allocator_type::propagate_on_container_move_assignment` holds true.
`propagate_on_container_swap`	The type holds true only if `Outer_traits::propagate_on_container_swap` holds true or `inner_allocator_type::propagate_on_container_swap` holds true.
`size_type`	This type is a synonym for the `size_type` associated with the allocator `Outer`.
`value_type`	This type is a synonym for the `value_type` associated with the allocator `Outer`.
`void_pointer`	This type is a synonym for the `void_pointer` associated with the allocator `Outer`.

Structs

Name	Description
scoped_allocator_adaptor::rebind Struct	Defines the type `Outer::rebind\<Other>::other` as a synonym for `scoped_allocator_adaptor\<Other, Inner...>`.

Methods

Name	Description
allocate	Allocates memory by using the `Outer` allocator.
construct	Constructs an object.
deallocate	Deallocates objects by using the outer allocator.
destroy	Destroys a specified object.
inner_allocator	Retrieves a reference to the stored object of type `inner_allocator_type`.
max_size	Determines the maximum number of objects that can be allocated by the outer allocator.

Name	Description
outer_allocator	Retrieves a reference to the stored object of type `outer_allocator_type`.
select_on_container_copy_construction	Creates a new `scoped_allocator_adaptor` object with each stored allocator object initialized by calling `select_on_container_copy_construction` for each corresponding allocator.

Operators

Operator	Description
operator=	
operator==	
operator!=	

Requirements

Header: <scoped_allocator>

Namespace: std

scoped_allocator_adaptor::allocate

Allocates memory by using the `Outer` allocator.

```
C++
```

```
pointer allocate(size_type count);pointer allocate(size_type count, const_void_pointer hint);
```

Parameters

count
The number of elements for which sufficient storage is to be allocated.

hint
A pointer that might assist the allocator object by locating the address of an object allocated prior to the request.

Return Value

The first member function returns `Outer_traits::allocate(outer_allocator(), count)`. The second member function returns `Outer_traits::allocate(outer_allocator(), count, hint)`.

scoped_allocator_adaptor::construct

Constructs an object.

```
C++
```

```
template <class Ty, class... Atypes>
void construct(Ty* ptr, Atypes&&... args);
```

```
template <class Ty1, class Ty2, class... Atypes1, class... Atypes2>
void construct(pair<Ty1, Ty2>* ptr, piecewise_construct_t,
    tuple<Atypes1&&...>
first, tuple<Atypes1&&...> second);

template <class Ty1, class Ty2>
void construct(pair<Ty1, Ty2>* ptr);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr,
    class Uy1&& first, class Uy2&& second);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr, const pair<Uy1, Uy2>& right);

template <class Ty1, class Ty2, class Uy1, class Uy2>
void construct(pair<Ty1, Ty2>* ptr, pair<Uy1, Uy2>&& right);
```

Parameters

ptr
A pointer to the memory location where the object is to be constructed.

args
A list of arguments.

first
An object of the first type in a pair.

second
An object of the second type in a pair.

right
An existing object to be moved or copied.

Remarks

The first method constructs the object at *ptr* by calling `Outermost_traits::construct(OUTERMOST(*this), ptr, xargs...)`, where `xargs...` is one of the following.

- If `uses_allocator<Ty, inner_allocator_type>` holds false, then `xargs...` is `args...`.
- If `uses_allocator<Ty, inner_allocator_type>` holds true, and `is_constructible<Ty, allocator_arg_t, inner_allocator_type, args...>` holds true, then `xargs...` is `allocator_arg, inner_allocator(), args...`.
- If `uses_allocator<Ty, inner_allocator_type>` holds true, and `is_constructible<Ty, args..., inner_allocator()>` holds true, then `xargs...` is `args..., inner_allocator()`.

The second method constructs the pair object at *ptr* by calling `Outermost_traits::construct(OUTERMOST(*this), &ptr->first, xargs...)`, where `xargs...` is `first...` modified as in the above list, and `Outermost_traits::construct(OUTERMOST(*this), &ptr->second, xargs...)`, where `xargs...` is `second...` modified as in the above list.

The third method behaves the same as `this->construct(ptr, piecewise_construct, tuple<>, tuple<>)`.

The fourth method behaves the same as `this->construct(ptr, piecewise_construct, forward_as_tuple(std::forward<Uy1>(first), forward_as_tuple(std::forward<Uy2>(second))`.

The fifth method behaves the same as `this->construct(ptr, piecewise_construct, forward_as_tuple(right.first), forward_as_tuple(right.second))`.

The sixth method behaves the same as `this->construct(ptr, piecewise_construct, forward_as_tuple(std::forward<Uy1>(right.first), forward_as_tuple(std::forward<Uy2>(right.second))`.

scoped_allocator_adaptor::deallocate

Deallocates objects by using the outer allocator.

```
C++

void deallocate(pointer ptr, size_type count);
```

Parameters

ptr
A pointer to the starting location of the objects to be deallocated.

count
The number of objects to deallocate.

scoped_allocator_adaptor::destroy

Destroys a specified object.

```
C++

template <class Ty>
void destroy(Ty* ptr)
```

Parameters

ptr
A pointer to the object to be destroyed.

Return Value

`Outermost_traits::destroy(OUTERMOST(*this), ptr)`

scoped_allocator_adaptor::inner_allocator

Retrieves a reference to the stored object of type `inner_allocator_type`.

```
C++

inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;
```

Return Value

A reference to the stored object of type `inner_allocator_type`.

scoped_allocator_adaptor::max_size

Determines the maximum number of objects that can be allocated by the outer allocator.

```
C++

size_type max_size();
```

Return Value

`Outer_traits::max_size(outer_allocator())`

scoped_allocator_adaptor::operator=

```
C++

scoped_allocator_adaptor& operator=(const scoped_allocator_adaptor&) = default;
scoped_allocator_adaptor& operator=(scoped_allocator_adaptor&&) = default;
```

scoped_allocator_adaptor::operator==

```
C++

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

scoped_allocator_adaptor::operator!=

```
C++

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

scoped_allocator_adaptor::outer_allocator

Retrieves a reference to the stored object of type `outer_allocator_type`.

```
C++

outer_allocator_type& outer_allocator() noexcept;
const outer_allocator_type& outer_allocator() const noexcept;
```

Return Value

A reference to the stored object of type `outer_allocator_type`.

scoped_allocator_adaptor::rebind Struct

Defines the type `Outer::rebind\<Other>::other` as a synonym for `scoped_allocator_adaptor\<Other, Inner...>`.

struct rebind{ typedef Other_traits::rebind<Other> Other_alloc; typedef scoped_allocator_adaptor<Other_alloc, Inner...> other; };

scoped_allocator_adaptor::scoped_allocator_adaptor Constructor

Constructs a `scoped_allocator_adaptor` object. Also includes a destructor.

```
C++
```

```
scoped_allocator_adaptor();

scoped_allocator_adaptor(const scoped_allocator_adaptor& right) noexcept;
template <class Outer2>
scoped_allocator_adaptor(
const scoped_allocator_adaptor<Outer2, Inner...>& right) noexcept;
template <class Outer2>
scoped_allocator_adaptor(
scoped_allocator_adaptor<Outer2, Inner...>&& right) noexcept;
template <class Outer2>
scoped_allocator_adaptor(Outer2&& al,
    const Inner&... rest) noexcept;

~scoped_allocator_adaptor();
```

Parameters

right
An existing `scoped_allocator_adaptor`.

al
An existing allocator to be used as the outer allocator.

rest
A list of allocators to be used as the inner allocators.

Remarks

The first constructor default constructs its stored allocator objects. Each of the next three constructors constructs its stored allocator objects from the corresponding objects in *right*. The last constructor constructs its stored allocator objects from the corresponding arguments in the argument list.

scoped_allocator_adaptor::select_on_container_copy_construction

Creates a new `scoped_allocator_adaptor` object with each stored allocator object initialized by calling `select_on_container_copy_construction` for each corresponding allocator.

```
C++
```

```
scoped_allocator_adaptor select_on_container_copy_construction();
```

Return Value

This method effectively returns `scoped_allocator_adaptor(Outer_traits::select_on_container_copy_construction(*this), inner_allocator().select_on_container_copy_construction())`. The result is a new `scoped_allocator_adaptor` object with each stored allocator object initialized by calling `al.select_on_container_copy_construction()` for the corresponding allocator *al*.

See also

Header Files Reference

<set>

Article • 12/07/2021

Defines the container class templates set and multiset and their supporting templates.

Requirements

Header: <set>

Namespace: std

> ⓘ Note
>
> The <set> library also uses the `#include <initializer_list>` statement.

Members

Operators

Set version	Multiset version	Description
operator!= (set)	operator!= (multiset)	Tests if the set or multiset object on the left side of the operator is not equal to the set or multiset object on the right side.
operator< (set)	operator< (multiset)	Tests if the set or multiset object on the left side of the operator is less than the set or multiset object on the right side.
operator<= (set)	operator<= (multiset)	Tests if the set or multiset object on the left side of the operator is less than or equal to the set or multiset object on the right side.
operator== (set)	operator== (multiset)	Tests if the set or multiset object on the left side of the operator is equal to the set or multiset object on the right side.
operator> (set)	operator> (multiset)	Tests if the set or multiset object on the left side of the operator is greater than the set or multiset object on the right side.
operator>= (set)	operator>= (multiset)	Tests if the set or multiset object on the left side of the operator is greater than or equal to the set or multiset object on the right side.

Specialized Template Functions

Set version	Multiset version	Description
swap	swap (multiset)	Exchanges the elements of two sets or multisets.

Classes

Name	Description
set Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered.
multiset Class	Used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<set>` functions

Article • 01/24/2023

swap (set)

Exchanges the elements of two sets.

```
C++
```

```
template <class Key, class Traits, class Allocator>
void swap(set<Key, Traits, Allocator>& left, set<Key, Traits, Allocator>&
right);
```

Parameters

right
The set providing the elements to be swapped, or the set whose elements are to be exchanged with those of the set *left*.

left
The set whose elements are to be exchanged with those of the set *right*.

Remarks

The template function is an algorithm specialized on the container class set to execute the member function `left.`swap(`right`). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

`template` < **classT**> **void swap**(**T&**, **T&**)

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class set::swap for an example of the use of the template version of `swap`.

swap (multiset)

Exchanges the elements of two multisets.

```
C++
```

```
template <class Key, class Traits, class Allocator>
void swap(multiset<Key, Traits, Allocator>& left, multiset<Key, Traits,
Allocator>& right);
```

Parameters

right
The multiset providing the elements to be swapped, or the multiset whose elements are to be exchanged with those of the multiset *left*.

left
The multiset whose elements are to be exchanged with those of the multiset *right*.

Remarks

The template function is an algorithm specialized on the container class multiset to execute the member function `left.`swap(`right`). This is an instance of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function

`template` < **classT**> **void swap**(**T&**, **T&**)

in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for the member class multiset::swapfor an example of the use of the template version of `swap`.

`<set>` operators

Article • 12/07/2021

operator!= (set)

Tests if the set object on the left side of the operator is not equal to the set object on the right side.

```
C++
```

```
bool operator!=(const set <Key, Traits, Allocator>& left, const set <Key,
Traits, Allocator>& right);
```

Parameters

left
An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the sets are not equal; `false` if sets are equal.

Remarks

The comparison between set objects is based on a pairwise comparison between their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// set_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 != s2 )
      cout << "The sets s1 and s2 are not equal." << endl;
   else
      cout << "The sets s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The sets s1 and s3 are not equal." << endl;
   else
      cout << "The sets s1 and s3 are equal." << endl;
}
/* Output:
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
*/
```

`operator<` (set)

Tests if the set object on the left side of the operator is less than the set object on the right side.

C++

```
bool operator<(const set <Key, Traits, Allocator>& left, const set <Key,
Traits, Allocator>& right);
```

Parameters

left
An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the set on the left side of the operator is strictly less than the set on the right side of the operator; otherwise `false`.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// set_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 < s2 )
      cout << "The set s1 is less than the set s2." << endl;
   else
      cout << "The set s1 is not less than the set s2." << endl;

   if ( s1 < s3 )
      cout << "The set s1 is less than the set s3." << endl;
   else
      cout << "The set s1 is not less than the set s3." << endl;
}
/* Output:
The set s1 is less than the set s2.
The set s1 is not less than the set s3.
*/
```

`operator<=` (set)

Tests if the set object on the left side of the operator is less than or equal to the set object on the right side.

```
C++
```

```
bool operator!<=(const set <Key, Traits, Allocator>& left, const set <Key,
Traits, Allocator>& right);
```

Parameters

left
An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the set on the left side of the operator is less than or equal to the set on the right side of the operator; otherwise `false`.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

```
C++
```

```
// set_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3, s4;
   int i;
```

```
   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 <= s2 )
      cout << "Set s1 is less than or equal to the set s2." << endl;
   else
      cout << "The set s1 is greater than the set s2." << endl;

   if ( s1 <= s3 )
      cout << "Set s1 is less than or equal to the set s3." << endl;
   else
      cout << "The set s1 is greater than the set s3." << endl;

   if ( s1 <= s4 )
      cout << "Set s1 is less than or equal to the set s4." << endl;
   else
      cout << "The set s1 is greater than the set s4." << endl;
}
```

Output

```
Set s1 is less than or equal to the set s2.
The set s1 is greater than the set s3.
Set s1 is less than or equal to the set s4.
```

operator== (set)

Tests if the set object on the left side of the operator is equal to the set object on the right side.

C++

```
bool operator!==(const set <Key, Traits, Allocator>& left, const set <Key,
Traits, Allocator>& right);
```

Parameters

left

An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the set on the left side of the operator is equal to the set on the right side of the operator; otherwise `false`.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. Two sets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// set_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The sets s1 and s2 are equal." << endl;
   else
      cout << "The sets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The sets s1 and s3 are equal." << endl;
   else
      cout << "The sets s1 and s3 are not equal." << endl;
}
```

Output

```
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
```

`operator>` (set)

Tests if the set object on the left side of the operator is greater than the set object on the right side.

C++

```
bool operator>(const set <Key, Traits, Allocator>& left, const set <Key,
Traits, Allocator>& right);
```

Parameters

left
An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the set on the left side of the operator is greater than the set on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// set_op_gt.cpp
// compile with: /EHsc
#include <set>
```

```cpp
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 > s2 )
      cout << "The set s1 is greater than the set s2." << endl;
   else
      cout << "The set s1 is not greater than the set s2." << endl;

   if ( s1 > s3 )
      cout << "The set s1 is greater than the set s3." << endl;
   else
      cout << "The set s1 is not greater than the set s3." << endl;
}
/* Output:
The set s1 is not greater than the set s2.
The set s1 is greater than the set s3.
*/
```

`operator>=` (set)

Tests if the set object on the left side of the operator is greater than or equal to the set object on the right side.

C++

```cpp
bool operator!>=(const set <Key, Traits, Allocator>& left, const set <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `set`.

right
An object of type `set`.

Return Value

`true` if the set on the left side of the operator is greater than or equal to the set on the right side of the list; otherwise `false`.

Remarks

The comparison between set objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// set_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 >= s2 )
      cout << "Set s1 is greater than or equal to set s2." << endl;
   else
      cout << "The set s1 is less than the set s2." << endl;

   if ( s1 >= s3 )
      cout << "Set s1 is greater than or equal to set s3." << endl;
   else
      cout << "The set s1 is less than the set s3." << endl;

   if ( s1 >= s4 )
      cout << "Set s1 is greater than or equal to set s4." << endl;
   else
      cout << "The set s1 is less than the set s4." << endl;
}
```

Output

```
The set s1 is less than the set s2.
Set s1 is greater than or equal to set s3.
Set s1 is greater than or equal to set s4.
```

operator!= (multiset)

Tests if the multiset object on the left side of the operator is not equal to the multiset object on the right side.

C++

```
bool operator!=(const multiset <Key, Traits, Allocator>& left, const
multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the sets or multisets are not equal; **`false`** if sets or multisets are equal.

Remarks

The comparison between multiset objects is based on a pairwise comparison between their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```cpp
// multiset_op_ne.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 != s2 )
      cout << "The multisets s1 and s2 are not equal." << endl;
   else
      cout << "The multisets s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The multisets s1 and s3 are not equal." << endl;
   else
      cout << "The multisets s1 and s3 are equal." << endl;
}
```

Output

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

`operator<` (multiset)

Tests if the multiset object on the left side of the operator is less than the multiset object on the right side.

C++

```cpp
bool operator<(const multiset <Key, Traits, Allocator>& left, const multiset
<Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the multiset on the left side of the operator is strictly less than the multiset on the right side of the operator; otherwise `false`.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multiset_op_lt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 < s2 )
      cout << "The multiset s1 is less than "
           << "the multiset s2." << endl;
   else
      cout << "The multiset s1 is not less than "
           << "the multiset s2." << endl;

   if ( s1 < s3 )
```

```
      cout << "The multiset s1 is less than "
           << "the multiset s3." << endl;
   else
      cout << "The multiset s1 is not less than "
           << "the multiset s3." << endl;
}
```

Output

```
The multiset s1 is less than the multiset s2.
The multiset s1 is not less than the multiset s3.
```

`operator<=` (multiset)

Tests if the multiset object on the left side of the operator is less than or equal to the multiset object on the right side.

C++

```
bool operator!<=(const multiset <Key, Traits, Allocator>& left, const
multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the multiset on the left side of the operator is less than or equal to the multiset on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multiset_op_le.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 <= s2 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s2." << endl;
   else
      cout << "The multiset s1 is greater than "
           << "the multiset s2." << endl;

   if ( s1 <= s3 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s3." << endl;
   else
      cout << "The multiset s1 is greater than "
           << "the multiset s3." << endl;

   if ( s1 <= s4 )
      cout << "The multiset s1 is less than "
           << "or equal to the multiset s4." << endl;
   else
      cout << "The multiset s1 is greater than "
           << "the multiset s4." << endl;
}
```

Output

```
The multiset s1 is less than or equal to the multiset s2.
The multiset s1 is greater than the multiset s3.
The multiset s1 is less than or equal to the multiset s4.
```

operator== (multiset)

Tests if the multiset object on the left side of the operator is equal to the multiset object on the right side.

```
C++
```

```
bool operator!==(const multiset <Key, Traits, Allocator>& left, const
multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the multiset on the left side of the operator is equal to the multiset on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. Two sets or multisets are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// multiset_op_eq.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;
```

```
   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i );
   }

   if ( s1 == s2 )
      cout << "The multisets s1 and s2 are equal." << endl;
   else
      cout << "The multisets s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The multisets s1 and s3 are equal." << endl;
   else
      cout << "The multisets s1 and s3 are not equal." << endl;
}
```

Output

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

`operator>` (multiset)

Tests if the multiset object on the left side of the operator is greater than the multiset object on the right side.

C++

```
bool operator>(const multiset <Key, Traits, Allocator>& left, const multiset
<Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the multiset on the left side of the operator is greater than the multiset on the right side of the operator; otherwise `false`.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater-than relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multiset_op_gt.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
   }

   if ( s1 > s2 )
      cout << "The multiset s1 is greater than "
           << "the multiset s2." << endl;
   else
      cout << "The multiset s1 is not greater "
           << "than the multiset s2." << endl;

   if ( s1 > s3 )
      cout << "The multiset s1 is greater than "
           << "the multiset s3." << endl;
   else
      cout << "The multiset s1 is not greater than "
           << "the multiset s3." << endl;
}
```

Output

```
The multiset s1 is not greater than the multiset s2.
The multiset s1 is greater than the multiset s3.
```

`operator>=` (multiset)

Tests if the multiset object on the left side of the operator is greater than or equal to the multiset object on the right side.

C++

```
bool operator!>=(const multiset <Key, Traits, Allocator>& left, const
multiset <Key, Traits, Allocator>& right);
```

Parameters

left
An object of type `multiset`.

right
An object of type `multiset`.

Return Value

`true` if the multiset on the left side of the operator is greater than or equal to the multiset on the right side of the list; otherwise **`false`**.

Remarks

The comparison between multiset objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// multiset_op_ge.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   multiset <int> s1, s2, s3, s4;
   int i;

   for ( i = 0 ; i < 3 ; i++ )
   {
      s1.insert ( i );
      s2.insert ( i * i );
      s3.insert ( i - 1 );
      s4.insert ( i );
   }

   if ( s1 >= s2 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s2." << endl;
   else
      cout << "The multiset s1 is less than "
           << "the multiset s2." << endl;

   if ( s1 >= s3 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s3." << endl;
   else
      cout << "The multiset s1 is less than "
           << "the multiset s3." << endl;

   if ( s1 >= s4 )
      cout << "The multiset s1 is greater than "
           << "or equal to the multiset s4." << endl;
   else
      cout << "The multiset s1 is less than "
           << "the multiset s4." << endl;
}
```

Output

```
The multiset s1 is less than the multiset s2.
The multiset s1 is greater than or equal to the multiset s3.
The multiset s1 is greater than or equal to the multiset s4.
```

`set` Class

Article • 08/17/2021

The C++ Standard Library container class `set` is used to store and retrieve data from a collection. The values of the elements in the `set` are unique and serve as the key values according to which the data is automatically ordered. The value of an element in a `set` may not be changed directly. Instead, you must delete old values and insert elements with new values.

Syntax

C++

```
template <class Key,
    class Traits=less<Key>,
    class Allocator=allocator<Key>>
class set
```

Parameters

`Key`

The element data type to be stored in the set.

`Traits`

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the set. This argument is optional, and the binary predicate `less <Key>` is the default value.

In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the set's allocation and deallocation of memory. This argument is optional, and the default value is `allocator<Key>`.

Remarks

A C++ Standard Library set is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Also, it's a simple associative container because its element values are its key values.
- Reversible, because it provides a bidirectional iterator to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Unique in the sense that each of its elements must have a unique key. Since set is also a simple associative container, its elements are also unique.

A set is also described as a class template because the functionality it provides is generic and independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations are efficient, doing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that pointed at the removed elements.

The set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a set are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure. If values need to be attached to a list of unique key words, then a map would be an appropriate structure to contain this data. If instead the keys aren't unique, then a multimap would be the container of choice.

The set orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need to be merely less than comparable to establish this order so that, given any two elements, it may be determined either that they're equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary

predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f(x,y)* is a function object that has two argument objects *x* and *y* and a return value of `true` or `false`. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f x,y*) and *f(y,x)* are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

The iterator provided by the set class is a bidirectional iterator, but the class member functions insert and set have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it's enough to be able to talk meaningfully about a range of iterators [`First`, `Last`) in the context of the class's member functions.

Constructors

Name	Description
set	Constructs a set that is empty or that is a copy of all or part of some other set.

Typedefs

Name	Description
allocator_type	A type that represents the `allocator` class for the set object.
const_iterator	A type that provides a bidirectional iterator that can read a `const` element in the set.
const_pointer	A type that provides a pointer to a `const` element in a set.

Name	Description
const_reference	A type that provides a reference to a `const` element stored in a set for reading and doing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the set.
difference_type	A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a set.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.
key_type	The type describes an object stored as an element of a set in its capacity as sort key.
pointer	A type that provides a pointer to an element in a set.
reference	A type that provides a reference to an element stored in a set.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed set.
size_type	An unsigned integer type that can represent the number of elements in a set.
value_compare	The type that provides a function object that can compare two elements to determine their relative order in the set.
value_type	The type describes an object stored as an element of a set in its capacity as a value.

Functions

Name	Description
begin	Returns an iterator that addresses the first element in the `set`.
cbegin	Returns a const iterator that addresses the first element in the `set`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `set`.
clear	Erases all the elements of a `set`.
contains C++20	Check if there's an element with the specified key in the `set`.

Name	Description
count	Returns the number of elements in a `set` whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed `set`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `set`.
emplace	Inserts an element constructed in place into a `set`.
emplace_hint	Inserts an element constructed in place into a `set`, with a placement hint.
empty	Tests if a `set` is empty.
end	Returns an iterator that addresses the location succeeding the last element in a `set`.
equal_range	Returns a pair of iterators respectively to the first element in a `set` with a key that is greater than a specified key and to the first element in the `set` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a `set` that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the `set`.
insert	Inserts an element or a range of elements into a `set`.
key_comp	Retrieves a copy of the comparison object used to order keys in a `set`.
lower_bound	Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `set`.
rbegin	Returns an iterator addressing the first element in a reversed `set`.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed `set`.
size	Returns the number of elements in the `set`.
swap	Exchanges the elements of two `set`s.
upper_bound	Returns an iterator to the first element in a `set` with a key that is greater than a specified key.

Name	Description
value_comp	Retrieves a copy of the comparison object used to order element values in a `set`.

Operators

Name	Description
operator=	Replaces the elements of a set with a copy of another set.

`allocator_type`

A type that represents the allocator class for the set object.

```
C++
```

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter Allocator.

Returns the function object that a multiset uses to order its elements, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the set Class topic.

Example

See the example for get_allocator for an example that uses `allocator_type`.

`begin`

Returns an iterator that addresses the first element in the set.

```
C++
```

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the set or the location succeeding an empty set.

Remarks

If the return value of `begin` is assigned to a `const_iterator`, the elements in the set object can't be modified. If the return value of `begin` is assigned to an `iterator`, the elements in the set object can be modified.

Example

C++

```
// set_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::const_iterator s1_cIter;

   s1.insert( 1 );
   s1.insert( 2 );
   s1.insert( 3 );

   s1_Iter = s1.begin( );
   cout << "The first element of s1 is " << *s1_Iter << endl;

   s1_Iter = s1.begin( );
   s1.erase( s1_Iter );

   // The following 2 lines would err because the iterator is const
   // s1_cIter = s1.begin( );
   // s1.erase( s1_cIter );

   s1_cIter = s1.begin( );
   cout << "The first element of s1 is now " << *s1_cIter << endl;
}
```

Output

```
The first element of s1 is 1
The first element of s1 is now 2
```

cbegin

Returns a `const` iterator that addresses the first element in the range.

```
C++
```

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++
```

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

C++

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

C++

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`clear`

Erases all the elements of a set.

C++

```
void clear();
```

Example

C++

```
// set_clear.cpp
// compile with: /EHsc
```

```cpp
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 1 );
   s1.insert( 2 );

   cout << "The size of the set is initially " << s1.size( )
        << "." << endl;

   s1.clear( );
   cout << "The size of the set after clearing is "
        << s1.size( ) << "." << endl;
}
```

Output

```
The size of the set is initially 2.
The size of the set after clearing is 0.
```

`const_iterator`

A type that provides a bidirectional iterator that can read a **`const`** element in the set.

C++

```cpp
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

Example

See the example for begin for an example that uses `const_iterator`.

`const_pointer`

A type that provides a pointer to a **`const`** element in a set.

C++

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

In most cases, a const_iterator should be used to access the elements in a const set object.

`const_reference`

A type that provides a reference to a **`const`** element stored in a set for reading and doing **`const`** operations.

C++

```
typedef typename allocator_type::const_reference const_reference;
```

Example

C++

```
// set_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 10 );
   s1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *s1.begin( );

   cout << "The first element in the set is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference can't be used to modify the set
```

```
    // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the set is 10.
```

`const_reverse_iterator`

A type that provides a bidirectional iterator that can read any `const` element in the set.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the set in reverse.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

`contains`

Check if there's an element with the specified key in the `set`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

K

The type of the key.

key

The element's key value to look for.

Return Value

`true` if the element is found in the `set`; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent. See Heterogeneous lookup in associative containers for more information.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <set>
#include <iostream>

int main()
{
    std::set<int> theSet = {1, 2};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theSet.contains(2) << '\n';
    std::cout << theSet.contains(3) << '\n';

    return 0;
}
```

Output

```
true
false
```

count

Returns the number of elements in a set whose key matches a parameter-specified key.

C++

```
size_type count(const Key& key) const;
```

Parameters

key

The key of the elements to be matched from the set.

Return Value

1 if the set contains an element whose sort key matches the parameter key. 0 if the set doesn't contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[lower_bound(*key*), upper_bound(*key*)).

Example

The following example demonstrates the use of the `set::count` member function.

C++

```
// set_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    set<int> s1;
    set<int>::size_type i;

    s1.insert(1);
    s1.insert(1);

    // Keys must be unique in set, so duplicates are ignored
    i = s1.count(1);
    cout << "The number of elements in s1 with a sort key of 1 is: "
         << i << "." << endl;

    i = s1.count(2);
```

```
    cout << "The number of elements in s1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in s1 with a sort key of 1 is: 1.
The number of elements in s1 with a sort key of 2 is: 0.
```

crbegin

Returns a const iterator addressing the first element in a reversed set.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

`crbegin` is used with a reversed set just as begin is used with a set.

With the return value of `crbegin`, the set object can't be modified.

Example

C++

```
// set_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
```

```
    s1.insert( 20 );
    s1.insert( 30 );

    s1_crIter = s1.crbegin( );
    cout << "The first element in the reversed set is "
         << *s1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed set is 30.
```

crend

Returns a const iterator that addresses the location succeeding the last element in a reversed set.

C++

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed set (the location that had preceded the first element in the unreversed set).

Remarks

`crend` is used with a reversed set just as end is used with a set.

With the return value of `crend`, the set object can't be modified. The value returned by `crend` shouldn't be dereferenced.

`crend` can be used to test to whether a reverse iterator has reached the end of its set.

Example

C++

```
// set_crend.cpp
// compile with: /EHsc
```

```
#include <set>
#include <iostream>

int main() {
   using namespace std;
   set <int> s1;
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_crIter = s1.crend( );
   s1_crIter--;
   cout << "The last element in the reversed set is "
        << *s1_crIter << "." << endl;
}
```

`difference_type`

A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.

C++

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range *[first, last)* between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

C++

```cpp
// set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   set <int> s1;
   set <int>::iterator s1_Iter, s1_bIter, s1_eIter;

   s1.insert( 20 );
   s1.insert( 10 );
   s1.insert( 20 );   // won't insert as set elements are unique

   s1_bIter = s1.begin( );
   s1_eIter = s1.end( );

   set <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( s1_bIter, s1_eIter, 5 );
   df_typ10 = count( s1_bIter, s1_eIter, 10 );
   df_typ20 = count( s1_bIter, s1_eIter, 20 );

   // the keys, and hence the elements of a set are unique,
   // so there's at most one of a given value
   cout << "The number '5' occurs " << df_typ5
        << " times in set s1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in set s1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in set s1.\n";

   // count the number of elements in a set
   set <int>::difference_type  df_count = 0;
   s1_Iter = s1.begin( );
   while ( s1_Iter != s1_eIter)
   {
      df_count++;
      s1_Iter++;
   }

   cout << "The number of elements in the set s1 is: "
        << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in set s1.
The number '10' occurs 1 times in set s1.
```

```
The number '20' occurs 1 times in set s1.
The number of elements in the set s1 is: 2.
```

`emplace`

Inserts an element constructed in place (no copy or move operations are performed).

C++

```
template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);
```

Parameters

`args`

The arguments forwarded to construct an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.

Return Value

A pair whose bool component returns true if an insertion was made, and false if the map already contained an element whose value had an equivalent value in the ordering. The iterator component of the return value pair returns the address where a new element was inserted (if the bool component is true) or where the element was already located (if the bool component is false).

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state isn't modified.

Example

C++

```
// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
```

```cpp
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    set<string> s1;

    auto ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
            << endl << "  The existing element is (" << *ret.first << ")"
            << endl;
        cout << "set not modified" << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;

    ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
            << endl << "  The existing element is (" << *ret.first << ")"
            << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;
}
```

`emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
```

C++

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

args

The arguments forwarded to construct an element to be inserted into the set unless the set already contains that element or, more generally, unless it already contains an element whose value is equivalently ordered.

where

The place to start searching for the correct point of insertion. (If that point immediately precedes *where*, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state isn't modified.

Example

C++

```
// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;
```

```
template <typename S> void print(const S& s) {
    cout << s.size() << " elements: " << endl;

    for (const auto& p : s) {
        cout << "(" << p <<  ") ";
    }

    cout << endl;
}

int main()
{
    set<string> s1;

    // Emplace some test data
    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmine");

    cout << "set starting data: ";
    print(s1);
    cout << endl;

    // Emplace with hint
    // s1.end() should be the "next" element after this emplacement
    s1.emplace_hint(s1.end(), "Doug");

    cout << "set modified, now contains ";
    print(s1);
    cout << endl;
}
```

`empty`

Tests if a set is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the set is empty; `false` if the set is nonempty.

Example

C++

```cpp
// set_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2;
   s1.insert ( 1 );

   if ( s1.empty( ) )
      cout << "The set s1 is empty." << endl;
   else
      cout << "The set s1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The set s2 is empty." << endl;
   else
      cout << "The set s2 is not empty." << endl;
}
```

Output

```
The set s1 is not empty.
The set s2 is empty.
```

end

Returns the past-the-end iterator.

C++

```cpp
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the set is empty, then `set::end() == set::begin()`.

Remarks

`end` is used to test whether an iterator has passed the end of its set.

The value returned by `end` shouldn't be dereferenced.

For a code example, see set::find.

`equal_range`

Returns a pair of iterators respectively to the first element in a set with a key that is greater than or equal to a specified key and to the first element in the set with a key that is greater than the key.

```
C++
```

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the set being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **first**, and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second**, and to dereference the upper bound iterator, use *(`pr`. **second**).

Example

```
C++
```

```
// set_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
```

```
   using namespace std;
   typedef set<int, less< int > > IntSet;
   IntSet s1;
   set <int, less< int > > :: const_iterator s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
   p1 = s1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   s1_RcIter = s1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *s1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = s1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == s1.end( ) ) && ( p2.second == s1.end( ) ) )
      cout << "The set s1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of set s1 with a key >= 40 is: "
           << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the set s1 is: 30.
The lower bound of the element with a key of 20 in the set s1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The set s1 doesn't have an element with a key less than 40.
```

erase

Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.

```
C++
```

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the set if no such element exists.

For the third member function, returns the number of elements that have been removed from the set.

Example

```
C++
```

```
// set_erase.cpp
// compile with: /EHsc
```

```
#include <set>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions

using namespace std;

using myset = set<string>;

void printset(const myset& s) {
    for (const auto& iter : s) {
        cout << " [" << iter << "]";
    }
    cout << endl << "size() == " << s.size() << endl << endl;
}

int main()
{
    myset s1;

    // Fill in some data to test with, one at a time
    s1.insert("Bob");
    s1.insert("Robert");
    s1.insert("Bert");
    s1.insert("Rob");
    s1.insert("Bobby");

    cout << "Starting data of set s1 is:" << endl;
    printset(s1);
    // The 1st member function removes an element at a given position
    s1.erase(next(s1.begin()));
    cout << "After the 2nd element is deleted, the set s1 is:" << endl;
    printset(s1);

    // Fill in some data to test with, one at a time, using an initializer
list
    myset s2{ "meow", "hiss", "purr", "growl", "yowl" };

    cout << "Starting data of set s2 is:" << endl;
    printset(s2);
    // The 2nd member function removes elements
    // in the range [First, Last)
    s2.erase(next(s2.begin()), prev(s2.end()));
    cout << "After the middle elements are deleted, the set s2 is:" << endl;
    printset(s2);

    myset s3;

    // Fill in some data to test with, one at a time, using emplace
    s3.emplace("C");
    s3.emplace("C#");
    s3.emplace("D");
    s3.emplace("D#");
    s3.emplace("E");
    s3.emplace("E#");
```

```
    s3.emplace("F");
    s3.emplace("F#");
    s3.emplace("G");
    s3.emplace("G#");
    s3.emplace("A");
    s3.emplace("A#");
    s3.emplace("B");

    cout << "Starting data of set s3 is:" << endl;
    printset(s3);
    // The 3rd member function removes elements with a given Key
    myset::size_type count = s3.erase("E#");
    // The 3rd member function also returns the number of elements removed
    cout << "The number of elements removed from s3 is: " << count << "." <<
endl;
    cout << "After the element with a key of \"E#\" is deleted, the set s3
is:" << endl;
    printset(s3);
}
```

`find`

Returns an iterator that refers to the location of an element in a set that has a key equivalent to a specified key.

C++

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key

The key value to be matched by the sort key of an element from the set being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the set (`set::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the set whose key is equivalent to the argument *key* under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the set object can't be modified. If the return value of `find` is assigned to an `iterator`, the set object can be modified

Example

C++

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    set<int> s1({ 40, 45 });
    cout << "The starting set s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
```

```
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate

    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified set s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

`get_allocator`

Returns a copy of the allocator object used to construct the set.

C++

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the set to manage memory, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the set Class topic.

Remarks

Allocators for the set class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes is sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```cpp
// set_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int>::allocator_type s1_Alloc;
   set <int>::allocator_type s2_Alloc;
   set <double>::allocator_type s3_Alloc;
   set <int>::allocator_type s4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   set <int> s1;
   set <int, allocator<int> > s2;
   set <double, allocator<double> > s3;

   s1_Alloc = s1.get_allocator( );
   s2_Alloc = s2.get_allocator( );
   s3_Alloc = s3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << s2.max_size( ) << "." << endl;

   cout << "\nThe number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << s3.max_size( ) <<  "." << endl;

   // The following line creates a set s4
   // with the allocator of multiset s1.
   set <int> s4( less<int>( ), s1_Alloc );

   s4_Alloc = s4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( s1_Alloc == s4_Alloc )
   {
      cout << "\nThe allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "\nThe allocators are not interchangeable."
           << endl;
   }
}
```

insert

Inserts an element or a range of elements into a set.

```
C++
```

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.

Where

The place to start searching for the correct point of insertion. (If that point immediately precedes *Where*, insertion can occur in amortized constant time instead of logarithmic time.)

`ValTy`

Template parameter that specifies the argument type that the set can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

`First`

The position of the first element to be copied.

`Last`

The position just beyond the last element to be copied.

`InputIterator`

Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

`IList`

The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose **`bool`** component is true if an insertion was made, and false if the set already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **`bool`** component is true, or to the existing element if the **`bool`** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the set or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state isn't modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the **`bool`** component, use `pr.second`. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and, for set, `set<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a set that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` doesn't get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `s.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `s`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the set.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see set::emplace and set::emplace_hint.

Example

C++

```
// set_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{

    // insert single values
    set<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
```

```
    s1.insert(20);

    cout << "The original set values of s1 are:" << endl;
    print(s1);

    // intentionally attempt a duplicate, single element
    auto ret = s1.insert(1);
    if (!ret.second){
        auto elem = *ret.first;
        cout << "Insert failed, element with value 1 already exists."
            << endl << "  The existing element is (" << elem << ")"
            << endl;
    }
    else{
        cout << "The modified set values of s1 are:" << endl;
        print(s1);
    }
    cout << endl;

    // single element, with hint
    s1.insert(s1.end(), 30);
    cout << "The modified set values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // The templatized version inserting a jumbled range
    set<int> s2;
    vector<int> v;
    v.push_back(43);
    v.push_back(294);
    v.push_back(41);
    v.push_back(330);
    v.push_back(42);
    v.push_back(45);

    cout << "Inserting the following vector data into s2:" << endl;
    print(v);

    s2.insert(v.begin(), v.end());

    cout << "The modified set values of s2 are:" << endl;
    print(s2);
    cout << endl;

    // The templatized versions move-constructing elements
    set<string>  s3;
    string str1("blue"), str2("green");

    // single element
    s3.insert(move(str1));
    cout << "After the first move insertion, s3 contains:" << endl;
    print(s3);

    // single element with hint
    s3.insert(s3.end(), move(str2));
```

```
    cout << "After the second move insertion, s3 contains:" << endl;
    print(s3);
    cout << endl;

    set<int> s4;
    // Insert the elements from an initializer_list
    s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
    cout << "After initializer_list insertion, s4 contains:" << endl;
    print(s4);
    cout << endl;
}
```

`iterator`

A type that provides a constant bidirectional iterator that can read any element in a set.

C++

```
typedef implementation-defined iterator;
```

Example

See the example for begin for an example of how to declare and use an `iterator`.

`key_comp`

Retrieves a copy of the comparison object used to order keys in a set.

C++

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits`, see the set Class topic.

Remarks

The stored object defines the member function:

bool operator()(const Key& `_xVal`**, const Key&** `_yVal`**);**

which returns **`true`** if `_xVal` precedes and isn't equal to `_yVal` in the sort order.

Both key_compare and value_compare are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

C++

```
// set_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set<int, less<int> >::key_compare kc1 = s1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of s1."
           << endl;
   }

   set <int, greater<int> > s2;
   set<int, greater<int> >::key_compare kc2 = s2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if(result2 == true)
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of s2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of s2."
           << endl;
```

```
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of s2.
```

key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.

C++

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits`, see the set Class topic.

Both `key_compare` and value_compare are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

key_type

A type that describes an object stored as an element of a set in its capacity as sort key.

C++

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the set Class topic.

Both `key_type` and value_type are synonyms for the template parameter `Key`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

See the example for value_type for an example of how to declare and use `key_type`.

`lower_bound`

Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.

C++

```
const_iterator lower_bound(const Key& key) const;

iterator lower_bound(const Key& key);
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

C++

```
// set_lower_bound.cpp
// compile with: /EHsc
```

```
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.lower_bound( 20 );
   cout << "The element of set s1 with a key of 20 is: "
        << *s1_RcIter << "." << endl;

   s1_RcIter = s1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of set s1 with a key of 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator that addresses the location
   s1_AcIter = s1.end( );
   s1_AcIter--;
   s1_RcIter = s1.lower_bound( *s1_AcIter );
   cout << "The element of s1 with a key matching "
        << "that of the last element is: "
        << *s1_RcIter << "." << endl;
}
```

Output

```
The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

max_size

Returns the maximum length of the set.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the set.

Example

C++

```
// set_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::size_type i;

   i = s1.max_size( );
   cout << "The maximum possible length "
        << "of the set is " << i << "." << endl;
}
```

`operator=`

Replaces the elements of this `set` using elements from another `set`.

C++

```
set& operator=(const set& right);

set& operator=(set&& right);
```

Parameters

`right`

The `set` providing new elements to be assigned to this `set`.

Remarks

The first version of `operator=` uses an lvalue reference for *`right`*, to copy elements from *`right`* to this `set`.

The second version uses an rvalue reference for right. It moves elements from *`right`* to this `set`.

Any elements in this `set` before the operator function executes are discarded.

Example

C++

```
// set_operator_as.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
   {
   using namespace std;
   set<int> v1, v2, v3;
   set<int>::iterator iter;

   v1.insert(10);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
   }
```

`pointer`

A type that provides a pointer to an element in a set.

C++

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a set object.

`rbegin`

Returns an iterator addressing the first element in a reversed set.

C++

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

`rbegin` is used with a reversed set just as begin is used with a set.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the set object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the set object can be modified.

`rbegin` can be used to iterate through a set backwards.

Example

C++

```
// set_rbegin.cpp
// compile with: /EHsc
```

```
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rbegin( );
   cout << "The first element in the reversed set is "
        << *s1_rIter << "." << endl;

   // begin can be used to start an iteration
   // through a set in a forward order
   cout << "The set is:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration
   // through a set in a reverse order
   cout << "The reversed set is:";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
      cout << " " << *s1_rIter;
   cout << endl;

   // A set element can be erased by dereferencing to its key
   s1_rIter = s1.rbegin( );
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed set is "<< *s1_rIter << "." << endl;
}
```

Output

```
The first element in the reversed set is 30.
The set is: 10 20 30
The reversed set is: 30 20 10
After the erasure, the first element in the reversed set is 20.
```

reference

A type that provides a reference to an element stored in a set.

C++

```
typedef typename allocator_type::reference reference;
```

Example

C++

```
// set_reference.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 10 );
   s1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   const int &Ref1 = *s1.begin( );

   cout << "The first element in the set is "
        << Ref1 << "." << endl;
}
```

Output

```
The first element in the set is 10.
```

rend

Returns an iterator that addresses the location succeeding the last element in a reversed set.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed set (the location that had preceded the first element in the unreversed set).

Remarks

`rend` is used with a reversed set just as end is used with a set.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the set object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the set object can be modified. The value returned by `rend` shouldn't be dereferenced.

`rend` can be used to test to whether a reverse iterator has reached the end of its set.

Example

C++

```
// set_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rend( );
   s1_rIter--;
   cout << "The last element in the reversed set is "
        << *s1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // through a set in a forward order
   cout << "The set is: ";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << *s1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // through a set in a reverse order
   cout << "The reversed set is: ";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
```

```
      cout << *s1_rIter << " ";
   cout << "." << endl;

   s1_rIter = s1.rend( );
   s1_rIter--;
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rend( );
   --s1_rIter;
   cout << "After the erasure, the last element in the "
        << "reversed set is " << *s1_rIter << "." << endl;
}
```

`reverse_iterator`

A type that provides a bidirectional iterator that can read or modify an element in a reversed set.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the set in reverse.

Example

See the example for rbegin for an example of how to declare and use `reverse_iterator`.

`set`

Constructs a set that is empty or that is a copy of all or part of some other set.

C++

```
set();

explicit set(
    const Traits& Comp);

set(
    const Traits& Comp,
    const Allocator& Al);
```

```
set(
    const set& Right);

set(
    set&& Right);

set(
    initializer_list<Type> IList);

set(
    initializer_list<Type> IList,
    const Compare& Comp);

set(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp);

template <class InputIterator>
set(
    InputIterator First,
    InputIterator Last,
    const Traits& Comp,
    const Allocator& Al);
```

Parameters

`Al`

The storage allocator class to be used for this set object, which defaults to `Allocator`.

`Comp`

The comparison function of type `const Traits` used to order the elements in the set, which defaults to `Compare`.

`Rght`

The set of which the constructed set is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The initializer_list from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the set and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their sets.

All constructors store a function object of type `Traits` that is used to establish an order among the keys of the set and that can later be returned by calling key_comp.

The first three constructors specify an empty initial set, the second specifying the type of comparison function (`comp`) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (`al`) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the set `right`.

The next three constructors use an initializer_list to specify the elements.

The next three constructors copy the range [`first`, `last`) of a set with increasing explicitness in specifying the type of comparison function of class `Traits` and **`Allocator`**.

The eighth constructor specifies a copy of the set by moving `right`.

Example

C++

```
// set_set.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
```

```
int main()
{
    using namespace std;

    // Create an empty set s0 of key type integer
    set <int> s0;

    // Create an empty set s1 with the key comparison
    // function of less than, then insert 4 elements
    set <int, less<int> > s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(30);
    s1.insert(40);

    // Create an empty set s2 with the key comparison
    // function of less than, then insert 2 elements
    set <int, less<int> > s2;
    s2.insert(10);
    s2.insert(20);

    // Create a set s3 with the
    // allocator of set s1
    set <int>::allocator_type s1_Alloc;
    s1_Alloc = s1.get_allocator();
    set <int> s3(less<int>(), s1_Alloc);
    s3.insert(30);

    // Create a copy, set s4, of set s1
    set <int> s4(s1);

    // Create a set s5 by copying the range s1[ first,  last)
    set <int>::const_iterator s1_bcIter, s1_ecIter;
    s1_bcIter = s1.begin();
    s1_ecIter = s1.begin();
    s1_ecIter++;
    s1_ecIter++;
    set <int> s5(s1_bcIter, s1_ecIter);

    // Create a set s6 by copying the range s4[ first,  last)
    // and with the allocator of set s2
    set <int>::allocator_type s2_Alloc;
    s2_Alloc = s2.get_allocator();
    set <int> s6(s4.begin(), ++s4.begin(), less<int>(), s2_Alloc);

    cout << "s1 =";
    for (auto i : s1)
        cout << " " << i;
    cout << endl;

    cout << "s2 = " << *s2.begin() << " " << *++s2.begin() << endl;

    cout << "s3 =";
    for (auto i : s3)
```

```
        cout << " " << i;
    cout << endl;

    cout << "s4 =";
    for (auto i : s4)
        cout << " " << i;
    cout << endl;

    cout << "s5 =";
    for (auto i : s5)
        cout << " " << i;
    cout << endl;

    cout << "s6 =";
    for (auto i : s6)
        cout << " " << i;
    cout << endl;

    // Create a set by moving s5
    set<int> s7(move(s5));
    cout << "s7 =";
    for (auto i : s7)
        cout << " " << i;
    cout << endl;

    // Create a set with an initializer_list
    cout << "s8 =";
    set<int> s8{ { 1, 2, 3, 4 } };
    for (auto i : s8)
        cout << " " << i;
    cout << endl;

    cout << "s9 =";
    set<int> s9{ { 5, 6, 7, 8 }, less<int>() };
    for (auto i : s9)
        cout << " " << i;
    cout << endl;

    cout << "s10 =";
    set<int> s10{ { 10, 20, 30, 40 }, less<int>(), s9.get_allocator() };
    for (auto i : s10)
        cout << " " << i;
    cout << endl;
}
```

Output

```
s1 = 10 20 30 40s2 = 10 20s3 = 30s4 = 10 20 30 40s5 = 10 20s6 = 10s7 = 10
20s8 = 1 2 3 4s9 = 5 6 7 8s10 = 10 20 30 40
```

size

Returns the number of elements in the set.

C++

```
size_type size() const;
```

Return Value

The current length of the set.

Example

C++

```
// set_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: size_type i;

   s1.insert( 1 );
   i = s1.size( );
   cout << "The set length is " << i << "." << endl;

   s1.insert( 2 );
   i = s1.size( );
   cout << "The set length is now " << i << "." << endl;
}
```

Output

```
The set length is 1.
The set length is now 2.
```

`size_type`

An unsigned integer type that can represent the number of elements in a set.

C++

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for size for an example of how to declare and use `size_type`

swap

Exchanges the elements of two sets.

C++

```
void swap(
    set<Key, Traits, Allocator>& right);
```

Parameters

right
The argument set providing the elements to be swapped with the target set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two sets whose elements are being exchanged.

Example

C++

```
// set_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   set <int>::iterator s1_Iter;

   s1.insert( 10 );
   s1.insert( 20 );
```

```
   s1.insert( 30 );
   s2.insert( 100 );
   s2.insert( 200 );
   s3.insert( 300 );

   cout << "The original set s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   s1.swap( s2 );

   cout << "After swapping with s2, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( s1, s3 );

   cout << "After swapping with s3, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original set s1 is: 10 20 30.
After swapping with s2, list s1 is: 100 200.
After swapping with s3, list s1 is: 300.
```

upper_bound

Returns an iterator to the first element in a set that with a key that is greater than a specified key.

C++

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

key

The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a set that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

C++

```
// set_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.upper_bound( 20 );
   cout << "The first element of set s1 with a key greater "
        << "than 20 is: " << *s1_RcIter << "." << endl;

   s1_RcIter = s1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of set s1 with a key > 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator addressing the location
   s1_AcIter = s1.begin( );
   s1_RcIter = s1.upper_bound( *s1_AcIter );
   cout << "The first element of s1 with a key greater than"
        << endl << "that of the initial element of s1 is: "
```

```
        << *s1_RcIter << "." << endl;
}
```

Output

```
The first element of set s1 with a key greater than 20 is: 30.
The set s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than
that of the initial element of s1 is: 20.
```

`value_comp`

Retrieves a copy of the comparison object used to order element values in a set.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits`, see the set Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** `_xVal`, **const Key&** `_yVal`);

which returns **`true`** if `_xVal` precedes and isn't equal to `_yVal` in the sort order.

Both value_compare and key_compare are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

C++

```cpp
// set_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set <int, less<int> >::value_compare vc1 = s1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
           << "where vc1 is the function object of s1."
           << endl;
   }

   set <int, greater<int> > s2;
   set<int, greater<int> >::value_compare vc2 = s2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of s2."
           << endl;
   }
   else
   {
      cout << "vc2( 2,3 ) returns value of false, "
           << "where vc2 is the function object of s2."
           << endl;
   }
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of s1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of s2.
```

value_compare

A type that provides a function object that can compare two element values to determine their relative order in the set.

```
C++

typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits`, see the set Class topic.

Both key_compare and `value_compare` are synonyms for the template parameter `Traits`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

See the example for value_comp for an example of how to declare and use `value_compare`.

`value_type`

A type that describes an object stored as an element of a set in its capacity as a value.

```
C++

typedef Key value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the set Class topic.

Both key_type and `value_type` are synonyms for the template parameter `Key`. Both types are provided for the set and multiset classes, where they're identical, for compatibility with the map and multimap classes, where they're distinct.

Example

C++

```
// set_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;

   set <int>::value_type svt_Int;   // Declare value_type
   svt_Int = 10;            // Initialize value_type

   set <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type

   s1.insert( svt_Int );         // Insert value into s1
   s1.insert( skt_Int );         // Insert key into s1

   // A set accepts key_types or value_types as elements
   cout << "The set has elements:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++)
      cout << " " << *s1_Iter;
   cout << "." << endl;
}
```

Output

```
The set has elements: 10 20.
```

`multiset` Class

Article • 08/17/2021

The C++ Standard Library multiset class is used for the storage and retrieval of data from a collection in which the values of the elements contained need not be unique and in which they serve as the key values according to which the data is automatically ordered. The key value of an element in a `multiset` may not be changed directly. Instead, old values must be deleted and elements with new values inserted.

Syntax

C++

```
template <class Key, class Compare =less <Key>, class Allocator =allocator
<Key>>
class multiset
```

Parameters

`Key`

The element data type to be stored in the `multiset`.

`Compare`

The type that provides a function object that can compare two element values as sort keys to determine their relative order in the `multiset`. The binary predicate **less**<Key> is the default value.

In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the `multiset`'s allocation and deallocation of memory. The default value is `allocator<Key>`.

Remarks

The C++ Standard Library `multiset` class is:

- An associative container, which is a variable size container that supports the efficient retrieval of element values based on an associated key value.
- Reversible, because it provides bidirectional iterators to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Multiple in the sense that its elements don't need to have unique keys, so that one key value can have many element values associated with it.
- A simple associative container because its element values are its key values.
- A class template, because the functionality it provides is generic and so independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The iterator provided by the `multiset` class is a bidirectional iterator, but the class member functions insert and multiset have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`First`, `Last`) in the context of the class's member functions.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion, and removal. The member functions that explicitly support these operations are efficient, doing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had pointed at the removed elements.

The `multiset` should be the associative container of choice when the conditions associating the values with their keys are satisfies by the application. The elements of a `multiset` may be multiple and serve as their own sort keys, so keys aren't unique. A model for this type of structure is an ordered list of, say, words in which the words may occur more than once. Had multiple occurrences of the words not been allowed, then a

set would have been the appropriate container structure. If unique definitions were attached as values to the list of unique key words, then a map would be an appropriate structure to contain this data. If instead the definitions weren't unique, then a `multimap` would be the container of choice.

The `multiset` orders the sequence it controls by calling a stored function object of type *`Compare`*. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need be merely less than comparable to establish this order: so that, given any two elements, it may be determined either that they're equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*, *y*) is a function object that has two argument objects *x* and *y* and a return value of **`true`** or **`false`**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects x and y are defined to be equivalent when both *f*(*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14, you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. See Heterogeneous Lookup in Associative Containers for more information.

Constructors

Constructor	Description
multiset	Constructs a `multiset` that is empty or that is a copy of all or part of a specified `multiset`.

Typedefs

Type name	Description
allocator_type	A typedef for the `allocator` class for the `multiset` object.
const_iterator	A typedef for a bidirectional iterator that can read a **`const`** element in the `multiset`.

Type name	Description
const_pointer	A typedef for a pointer to a `const` element in a `multiset`.
const_reference	A typedef for a reference to a `const` element stored in a `multiset` for reading and doing `const` operations.
const_reverse_iterator	A typedef for a bidirectional iterator that can read any `const` element in the `multiset`.
difference_type	A signed integer typedef for the number of elements of a `multiset` in a range between elements pointed to by iterators.
iterator	A typedef for a bidirectional iterator that can read or modify any element in a `multiset`.
key_compare	A typedef for a function object that can compare two keys to determine the relative order of two elements in the `multiset`.
key_type	A typedef for a function object that can compare two sort keys to determine the relative order of two elements in the `multiset`.
pointer	A typedef for a pointer to an element in a `multiset`.
reference	A typedef for a reference to an element stored in a `multiset`.
reverse_iterator	A typedef for a bidirectional iterator that can read or modify an element in a reversed `multiset`.
size_type	An unsigned integer type that can represent the number of elements in a `multiset`.
value_compare	The typedef for a function object that can compare two elements as sort keys to determine their relative order in the `multiset`.
value_type	A typedef that describes an object stored as an element as a `multiset` in its capacity as a value.

Member functions

Member function	Description
begin	Returns an iterator that points to the first element in the `multiset`.
cbegin	Returns a const iterator that addresses the first element in the `multiset`.
cend	Returns a const iterator that addresses the location succeeding the last element in a `multiset`.

Member function	Description
clear	Erases all the elements of a `multiset`.
contains[C++20]	Check if there's an element with the specified key in the `multiset`.
count	Returns the number of elements in a `multiset` whose key matches the key specified as a parameter.
crbegin	Returns a const iterator addressing the first element in a reversed `multiset`.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed `multiset`.
emplace	Inserts an element constructed in place into a `multiset`.
emplace_hint	Inserts an element constructed in place into a `multiset`, with a placement hint.
empty	Tests if a `multiset` is empty.
end	Returns an iterator that points to the location after the last element in a `multiset`.
equal_range	Returns a pair of iterators. The first iterator in the pair points to the first element in a `multiset` with a key that is greater than a specified key. The second iterator in the pair points to first element in the `multiset` with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a `multiset` from specified positions or removes elements that match a specified key.
find	Returns an iterator that points to the first location of an element in a `multiset` that has a key equal to a specified key.
get_allocator	Returns a copy of the `allocator` object that is used to construct the `multiset`.
insert	Inserts an element or a range of elements into a `multiset`.
key_comp	Provides a function object that can compare two sort keys to determine the relative order of two elements in the `multiset`.
lower_bound	Returns an iterator to the first element in a `multiset` with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the `multiset`.
rbegin	Returns an iterator that points to the first element in a reversed `multiset`.
rend	Returns an iterator that points to the location succeeding the last element in a reversed `multiset`.

Member function	Description
size	Returns the number of elements in a `multiset`.
swap	Exchanges the elements of two `multiset`s.
upper_bound	Returns an iterator to the first element in a `multiset` with a key that is greater than a specified key.
value_comp	Retrieves a copy of the comparison object that is used to order element values in a `multiset`.

Operators

Operator	Description
operator=	Replaces the elements of a `multiset` with a copy of another `multiset`.

Requirements

Header: `<set>`

Namespace: `std`

`multiset::allocator_type`

A type that represents the allocator class for the `multiset` object

```
C++

typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the multiset Class topic.

Example

See the example for get_allocator for an example using `allocator_type`

`multiset::begin`

Returns an iterator addressing the first element in the `multiset`.

C++

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the `multiset` or the location succeeding an empty multiset.

Example

C++

```
// multiset_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::const_iterator ms1_cIter;

   ms1.insert( 1 );
   ms1.insert( 2 );
   ms1.insert( 3 );

   ms1_Iter = ms1.begin( );
   cout << "The first element of ms1 is " << *ms1_Iter << endl;

   ms1_Iter = ms1.begin( );
   ms1.erase( ms1_Iter );

   // The following 2 lines would err as the iterator is const
   // ms1_cIter = ms1.begin( );
   // ms1.erase( ms1_cIter );

   ms1_cIter = ms1.begin( );
   cout << "The first element of ms1 is now " << *ms1_cIter << endl;
}
```

Output

```
The first element of ms1 is 1
The first element of ms1 is now 2
```

`multiset::cbegin`

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` bidirectional-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`multiset::cend`

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A `const` bidirectional-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`multiset::clear`

Erases all the elements of a `multiset`.

```
C++
```

```
void clear();
```

Example

C++

```
// multiset_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 1 );
   ms1.insert( 2 );

   cout << "The size of the multiset is initially "
        << ms1.size( ) << "." << endl;

   ms1.clear( );
   cout << "The size of the multiset after clearing is "
        << ms1.size( ) << "." << endl;
}
```

Output

```
The size of the multiset is initially 2.
The size of the multiset after clearing is 0.
```

`multiset::const_iterator`

A type that provides a bidirectional iterator that can read a **`const`** element in the `multiset`.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

Example

See the example for begin for an example using `const_iterator`.

`multiset::const_pointer`

A type that provides a pointer to a **`const`** element in a `multiset`.

C++

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `multiset` object.

`multiset::const_reference`

A type that provides a reference to a **`const`** element stored in a `multiset` for reading and doing **`const`** operations.

C++

```
typedef typename allocator_type::const_reference const_reference;
```

Example

C++

```
// multiset_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
```

```
        << Ref1 << "." << endl;

    // The following line would cause an error because the
    // const_reference can't be used to modify the multiset
    // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the multiset is 10.
```

`multiset::const_reverse_iterator`

A type that provides a bidirectional iterator that can read any `const` element in the `multiset`.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the `multiset` in reverse.

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

`multiset::contains`

Check if there's an element with the specified key in the `multiset`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

K

The type of the key.

key

The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent. See Heterogeneous lookup in associative containers for more information.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <set>
#include <iostream>

int main()
{
    std::multiset<int> theMultiSet = {1, 2};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theMultiSet.contains(2) << '\n';
    std::cout << theMultiSet.contains(3) << '\n';

    return 0;
}
```

Output

```
true
false
```

`multiset::count`

Returns the number of elements in a `multiset` whose key matches a parameter-specified key.

```
C++
```

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the `multiset`.

Return Value

The number of elements in the `multiset` whose sort key matches the parameter key.

Remarks

The member function returns the number of elements *x* in the range

[lower_bound(*key*), upper_bound(*key*))

Example

The following example demonstrates the use of the `multiset`::count member function.

```
C++
```

```
// multiset_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    multiset<int> ms1;
    multiset<int>::size_type i;

    ms1.insert(1);
    ms1.insert(1);
    ms1.insert(2);

    // Elements don't need to be unique in multiset,
    // so duplicates are allowed and counted.
```

```
    i = ms1.count(1);
    cout << "The number of elements in ms1 with a sort key of 1 is: "
         << i << "." << endl;

    i = ms1.count(2);
    cout << "The number of elements in ms1 with a sort key of 2 is: "
         << i << "." << endl;

    i = ms1.count(3);
    cout << "The number of elements in ms1 with a sort key of 3 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in ms1 with a sort key of 1 is: 2.
The number of elements in ms1 with a sort key of 2 is: 1.
The number of elements in ms1 with a sort key of 3 is: 0.
```

`multiset::crbegin`

Returns a const iterator addressing the first element in a reversed multiset.

C++

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed multiset or addressing what had been the last element in the unreversed multiset.

Remarks

`crbegin` is used with a reversed multiset just as begin is used with a `multiset`.

With the return value of `crbegin`, the `multiset` object can't be modified.

`crbegin` can be used to iterate through a `multiset` backwards.

Example

C++

```cpp
// multiset_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_crIter = ms1.crbegin( );
   cout << "The first element in the reversed multiset is "
        << *ms1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed multiset is 30.
```

`multiset::crend`

Returns a const iterator that addresses the location succeeding the last element in a reversed multiset.

C++

```cpp
const_reverse_iterator crend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed multiset (the location that had preceded the first element in the unreversed multiset).

Remarks

`crend` is used with a reversed multiset just as end is used with a `multiset`.

With the return value of `crend`, the `multiset` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `multiset`.

The value returned by `crend` shouldn't be dereferenced.

Example

C++

```
// multiset_crend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   multiset <int> ms1;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_crIter = ms1.crend( ) ;
   ms1_crIter--;
   cout << "The last element in the reversed multiset is "
        << *ms1_crIter << "." << endl;
}
```

`multiset::difference_type`

A signed integer type that can be used to represent the number of elements of a `multiset` in a range between elements pointed to by iterators.

C++

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes

the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container like vector.

Example

C++

```
// multiset_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter, ms1_bIter, ms1_eIter;

   ms1.insert( 20 );
   ms1.insert( 10 );
   ms1.insert( 20 );

   ms1_bIter = ms1.begin( );
   ms1_eIter = ms1.end( );

   multiset <int>::difference_type   df_typ5, df_typ10, df_typ20;

   df_typ5 = count( ms1_bIter, ms1_eIter, 5 );
   df_typ10 = count( ms1_bIter, ms1_eIter, 10 );
   df_typ20 = count( ms1_bIter, ms1_eIter, 20 );

   // The keys, and hence the elements, of a multiset aren't unique
   cout << "The number '5' occurs " << df_typ5
        << " times in multiset ms1.\n";
   cout << "The number '10' occurs " << df_typ10
        << " times in multiset ms1.\n";
   cout << "The number '20' occurs " << df_typ20
        << " times in multiset ms1.\n";

   // Count the number of elements in a multiset
   multiset <int>::difference_type  df_count = 0;
   ms1_Iter = ms1.begin( );
   while ( ms1_Iter != ms1_eIter)
   {
```

```
        df_count++;
        ms1_Iter++;
    }

    cout << "The number of elements in the multiset ms1 is: "
         << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in multiset ms1.
The number '10' occurs 1 times in multiset ms1.
The number '20' occurs 2 times in multiset ms1.
The number of elements in the multiset ms1 is: 3.
```

`multiset::emplace`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace(Args&&... args);
```

Parameters

`args`
The arguments forwarded to construct an element to be inserted into the `multiset`.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state isn't modified.

Example

C++

```
// multiset_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    multiset<string> s1;

    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmine");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;

    s1.emplace("Bob");

    cout << "multiset modified, now contains ";
    print(s1);
    cout << endl;
}
```

`multiset::emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

`args`
The arguments forwarded to construct an element to be inserted into the `multiset`.

`where`
The place to start searching for the correct point of insertion. (If that point immediately precedes *`where`*, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During emplacement, if an exception is thrown, the container's state isn't modified.

For a code example, see set::emplace_hint.

`multiset::empty`

Tests if a `multiset` is empty.

C++

```
bool empty() const;
```

Return Value

`true` if the `multiset` is empty; **`false`** if the `multiset` is nonempty.

Example

C++

```
// multiset_empty.cpp
// compile with: /EHsc
```

```cpp
#include <set>
#include <iostream>

int main()
{
   using namespace std;
   multiset <int> ms1, ms2;
   ms1.insert ( 1 );

   if ( ms1.empty( ) )
      cout << "The multiset ms1 is empty." << endl;
   else
      cout << "The multiset ms1 is not empty." << endl;

   if ( ms2.empty( ) )
      cout << "The multiset ms2 is empty." << endl;
   else
      cout << "The multiset ms2 is not empty." << endl;
}
```

Output

```
The multiset ms1 is not empty.
The multiset ms2 is empty.
```

`multiset::end`

Returns the past-the-end iterator.

C++

```cpp
const_iterator end() const;

iterator end();
```

Return Value

The past-the-end iterator. If the `multiset` is empty, then `multiset::end() == multiset::begin()`.

Remarks

`end` is used to test whether an iterator has passed the end of its multiset.

The value returned by **`end`** shouldn't be dereferenced.

For a code example, see multiset::find.

`multiset::equal_range`

Returns a pair of iterators respectively to the first element in a `multiset` with a key that is greater than a specified key and to the first element in the `multiset` with a key that is equal to or greater than the key.

```
C++
```

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the `multiset` being searched.

Return Value

A pair of iterators such that the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **`first`**, and to dereference the lower bound iterator, use *(`pr`. **`first`**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **`second`**, and to dereference the upper bound iterator, use *(`pr`. **`second`**).

Example

```
C++
```

```
// multiset_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   typedef multiset<int, less<int> > IntSet;
```

```
    IntSet ms1;
    multiset <int> :: const_iterator ms1_RcIter;

    ms1.insert( 10 );
    ms1.insert( 20 );
    ms1.insert( 30 );

    pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
    p1 = ms1.equal_range( 20 );

    cout << "The upper bound of the element with "
         << "a key of 20 in the multiset ms1 is: "
         << *( p1.second ) << "." << endl;

    cout << "The lower bound of the element with "
         << "a key of 20 in the multiset ms1 is: "
         << *( p1.first ) << "." << endl;

    // Compare the upper_bound called directly
    ms1_RcIter = ms1.upper_bound( 20 );
    cout << "A direct call of upper_bound( 20 ) gives "
         << *ms1_RcIter << "," << endl
         << "matching the 2nd element of the pair"
         << " returned by equal_range( 20 )." << endl;

    p2 = ms1.equal_range( 40 );

    // If no match is found for the key,
    // both elements of the pair return end( )
    if ( ( p2.first == ms1.end( ) ) && ( p2.second == ms1.end( ) ) )
       cout << "The multiset ms1 doesn't have an element "
            << "with a key less than 40." << endl;
    else
       cout << "The element of multiset ms1 with a key >= 40 is: "
            << *( p1.first ) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the multiset ms1 is: 30.
The lower bound of the element with a key of 20 in the multiset ms1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The multiset ms1 doesn't have an element with a key less than 40.
```

`multiset::erase`

Removes an element or a range of elements in a `multiset` from specified positions or removes elements that match a specified key.

```
C++
```

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the `multiset` if no such element exists.

For the third member function, returns the number of elements that have been removed from the `multiset`.

Remarks

For a code example, see set::erase.

`multiset::find`

Returns an iterator that refers to the location of an element in a `multiset` that has a key equivalent to a specified key.

C++

```
iterator find(const Key& key);

const_iterator find(const Key& key) const;
```

Parameters

key
The key value to be matched by the sort key of an element from the `multiset` being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the `multiset` (`multiset::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the `multiset` whose key is equivalent to the argument *key* under a binary predicate that induces an ordering based on a less than comparability relation.

If the return value of `find` is assigned to a `const_iterator`, the `multiset` object can't be modified. If the return value of `find` is assigned to an `iterator`, the `multiset` object can be modified

Example

C++

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}
```

```
template <typename T> void print_collection(const T& t) {
    cout << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    multiset<int> s1({ 40, 45 });
    cout << "The starting multiset s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate

    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified multiset s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

`multiset::get_allocator`

Returns a copy of the allocator object used to construct the `multiset`.

```
C++
```

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the `multiset`.

Remarks

Allocators for the `multiset` class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

C++

```
// multiset_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int>::allocator_type ms1_Alloc;
   multiset <int>::allocator_type ms2_Alloc;
   multiset <double>::allocator_type ms3_Alloc;
   multiset <int>::allocator_type ms4_Alloc;

   // The following lines declare objects
   // that use the default allocator.
   multiset <int> ms1;
   multiset <int, allocator<int> > ms2;
   multiset <double, allocator<double> > ms3;

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms2.max_size( ) << "." << endl;

   cout << "The number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << ms3.max_size( ) <<  "." << endl;

   // The following lines create a multiset ms4
   // with the allocator of multiset ms1
   ms1_Alloc = ms1.get_allocator( );
   multiset <int> ms4( less<int>( ), ms1_Alloc );
```

```
   ms4_Alloc = ms4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated with the other
   if( ms1_Alloc == ms4_Alloc )
   {
      cout << "Allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "Allocators are not interchangeable."
           << endl;
   }
}
```

`multiset::insert`

Inserts an element or a range of elements into a `multiset`.

C++

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
```

```
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to be inserted into the `multiset`.

Where

The place to start searching for the correct point of insertion. (If that point immediately precedes *Where*, insertion can occur in amortized constant time instead of logarithmic time.)

ValTy

Template parameter that specifies the argument type that the `multiset` can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First

The position of the first element to be copied.

Last

The position just beyond the last element to be copied.

1

Template function argument that meets the requirements of an input_iterator_tag that points to elements of a type that can be used to construct 1 objects.

IList

The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the `multiset`.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the `multiset`.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown, the container's state isn't modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The value_type of a container is a typedef that belongs to the container, and, for set, `multiset<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a `multiset` that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `s.insert(v.begin(), v.end());` inserts all elements of `v` into `s`.

The initializer list member function (6) uses an initializer_list to copy elements into the `multiset`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see multiset::emplace and multiset::emplace_hint.

Example

C++

```
// multiset_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
```

```
    // insert single values
    multiset<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    s1.insert(20);

    cout << "The original multiset values of s1 are:" << endl;
    print(s1);

    // intentionally attempt a duplicate, single element
    s1.insert(1);
    cout << "The modified multiset values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // single element, with hint
    s1.insert(s1.end(), 30);
    cout << "The modified multiset values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // The templatized version inserting a jumbled range
    multiset<int> s2;
    vector<int> v;
    v.push_back(43);
    v.push_back(294);
    v.push_back(41);
    v.push_back(330);
    v.push_back(42);
    v.push_back(45);

    cout << "Inserting the following vector data into s2:" << endl;
    print(v);

    s2.insert(v.begin(), v.end());

    cout << "The modified multiset values of s2 are:" << endl;
    print(s2);
    cout << endl;

    // The templatized versions move-constructing elements
    multiset<string>  s3;
    string str1("blue"), str2("green");

    // single element
    s3.insert(move(str1));
    cout << "After the first move insertion, s3 contains:" << endl;
    print(s3);

    // single element with hint
    s3.insert(s3.end(), move(str2));
    cout << "After the second move insertion, s3 contains:" << endl;
    print(s3);
    cout << endl;
```

```
    multiset<int> s4;
    // Insert the elements from an initializer_list
    s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
    cout << "After initializer_list insertion, s4 contains:" << endl;
    print(s4);
    cout << endl;
}
```

`multiset::iterator`

A type that provides a constant bidirectional iterator that can read any element in a `multiset`.

C++

```
typedef implementation-defined iterator;
```

Example

See the example for begin for an example of how to declare and use an `iterator`.

`multiset::key_comp`

Retrieves a copy of the comparison object used to order keys in a `multiset`.

C++

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a `multiset` uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** *x*, **const Key&** *y*);

which returns true if *x* strictly precedes *y* in the sort order.

Both key_compare and value_compare are synonyms for the template parameter `Compare`. Both types are provided for the classes set and multiset, where they're identical, for compatibility with the classes map and multimap, where they're distinct.

Example

C++

```
// multiset_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::key_compare kc1 = ms1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of ms1."
           << endl;
   }

   multiset <int, greater<int> > ms2;
   multiset <int, greater<int> >::key_compare kc2 = ms2.key_comp( ) ;
   bool result2 = kc2( 2, 3 ) ;
   if( result2 == true )
   {
      cout << "kc2( 2,3 ) returns value of true, "
           << "where kc2 is the function object of ms2."
           << endl;
   }
   else
   {
      cout << "kc2( 2,3 ) returns value of false, "
           << "where kc2 is the function object of ms2."
           << endl;
```

```
   }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of ms2.
```

`multiset::key_compare`

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the `multiset`.

C++

```
typedef Compare key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

`multiset::key_type`

A type that provides a function object that can compare sort keys to determine the relative order of two elements in the `multiset`.

C++

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the multiset Class topic.

Example

See the example for value_type for an example of how to declare and use `key_type`.

`multiset::lower_bound`

Returns an iterator to the first element in a `multiset` with a key that is equal to or greater than a specified key.

C++

```
const_iterator lower_bound(const Key& key) const;

iterator lower_bound(const Key& key);
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the `multiset` being searched.

Return Value

An `iterator` or `const_iterator` that addresses the location of an element in a `multiset` that with a key that is equal to or greater than the argument key, or that addresses the location succeeding the last element in the `multiset` if no match is found for the key.

Example

C++

```
// multiset_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
```

```
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_RcIter = ms1.lower_bound( 20 );
   cout << "The element of multiset ms1 with a key of 20 is: "
        << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of multiset ms1 with a key of 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a dereferenced iterator addressing the location
   ms1_AcIter = ms1.end( );
   ms1_AcIter--;
   ms1_RcIter = ms1.lower_bound( *ms1_AcIter );
   cout << "The element of ms1 with a key matching "
        << "that of the last element is: "
        << *ms1_RcIter << "." << endl;
}
```

Output

```
The element of multiset ms1 with a key of 20 is: 20.
The multiset ms1 doesn't have an element with a key of 40.
The element of ms1 with a key matching that of the last element is: 30.
```

`multiset::max_size`

Returns the maximum length of the `multiset`.

C++

```
size_type max_size() const;
```

Return Value

The maximum possible length of the `multiset`.

Example

C++

```
// multiset_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::size_type i;

   i = ms1.max_size( );
   cout << "The maximum possible length "
        << "of the multiset is " << i << "." << endl;
}
```

`multiset::multiset`

Constructs a `multiset` that is empty or that is a copy of all or part of some other `multiset`.

C++

```
multiset();

explicit multiset (
    const Compare& Comp);

multiset (
    const Compare& Comp,
    const Allocator& Al);

multiset(
    const multiset& Right);

multiset(
    multiset&& Right);

multiset(
    initializer_list<Type> IList);

multiset(
    initializer_list<Type> IList,
    const Compare& Comp);

multiset(
```

```
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last,
    const Compare& Comp);

template <class InputIterator>
multiset (
    InputIterator First,
    InputIterator Last,
    const Compare& Comp,
    const Allocator& Al);
```

Parameters

Al

The storage allocator class to be used for this multiset object, which defaults to `Allocator`.

Comp

The comparison function of type `const Compare` used to order the elements in the `multiset`, which defaults to `Compare`.

Right

the `multiset` of which the constructed multiset is to be a copy.

First

The position of the first element in the range of elements to be copied.

Last

The position of the first element beyond the range of elements to be copied.

IList

The `initializer_list` from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the `multiset` and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their multiset.

All constructors store a function object of type Compare that is used to establish an order among the keys of the `multiset` and that can later be returned by calling key_comp.

The first three constructors specify an empty initial multiset, the second specifying the type of comparison function (*`Comp`*) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (*`Al`*) to be used. The keyword **`explicit`** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the `multiset` *`Right`*.

The fifth constructor specifies a copy of the `multiset` by moving *`Right`*.

The 6th, 7th, and 8th constructors specify an initializer_list from which to copy the elements.

The next three constructors copy the range `[First, Last)` of a `multiset` with increasing explicitness in specifying the type of comparison function and allocator.

Example

C++

```
// multiset_ctor.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    //multiset <int>::iterator ms1_Iter, ms2_Iter, ms3_Iter;
    multiset <int>::iterator ms4_Iter, ms5_Iter, ms6_Iter, ms7_Iter;

    // Create an empty multiset ms0 of key type integer
    multiset <int> ms0;

    // Create an empty multiset ms1 with the key comparison
    // function of less than, then insert 4 elements
    multiset <int, less<int> > ms1;
```

```
    ms1.insert(10);
    ms1.insert(20);
    ms1.insert(20);
    ms1.insert(40);

    // Create an empty multiset ms2 with the key comparison
    // function of greater than, then insert 2 elements
    multiset <int, less<int> > ms2;
    ms2.insert(10);
    ms2.insert(20);

    // Create a multiset ms3 with the
    // allocator of multiset ms1
    multiset <int>::allocator_type ms1_Alloc;
    ms1_Alloc = ms1.get_allocator();
    multiset <int> ms3(less<int>(), ms1_Alloc);
    ms3.insert(30);

    // Create a copy, multiset ms4, of multiset ms1
    multiset <int> ms4(ms1);

    // Create a multiset ms5 by copying the range ms1[ first,  last)
    multiset <int>::const_iterator ms1_bcIter, ms1_ecIter;
    ms1_bcIter = ms1.begin();
    ms1_ecIter = ms1.begin();
    ms1_ecIter++;
    ms1_ecIter++;
    multiset <int> ms5(ms1_bcIter, ms1_ecIter);

    // Create a multiset ms6 by copying the range ms4[ first,  last)
    // and with the allocator of multiset ms2
    multiset <int>::allocator_type ms2_Alloc;
    ms2_Alloc = ms2.get_allocator();
    multiset <int> ms6(ms4.begin(), ++ms4.begin(), less<int>(), ms2_Alloc);

    cout << "ms1 =";
    for (auto i : ms1)
        cout << " " << i;
    cout << endl;

    cout << "ms2 =";
    for (auto i : ms2)
        cout << " " << i;
   cout << endl;

   cout << "ms3 =";
   for (auto i : ms3)
       cout << " " << i;
    cout << endl;

    cout << "ms4 =";
    for (auto i : ms4)
        cout << " " << i;
    cout << endl;
```

```
    cout << "ms5 =";
    for (auto i : ms5)
        cout << " " << i;
    cout << endl;

    cout << "ms6 =";
    for (auto i : ms6)
        cout << " " << i;
    cout << endl;

    // Create a multiset by moving ms5
    multiset<int> ms7(move(ms5));
    cout << "ms7 =";
    for (auto i : ms7)
        cout << " " << i;
    cout << endl;

    // Create a multiset with an initializer_list
    multiset<int> ms8({1, 2, 3, 4});
    cout << "ms8=";
    for (auto i : ms8)
        cout << " " << i;
    cout << endl;
}
```

`multiset::operator=`

Replaces the elements of this `multiset` using elements from another `multiset`.

C++

```
multiset& operator=(const multiset& right);

multiset& operator=(multiset&& right);
```

Parameters

`Right`

The `multiset` from which elements are copied or moved.

Remarks

`operator=` copies or moves the elements in *`Right`* into this `multiset`, depending on the reference type (lvalue or rvalue) used. Elements that are in this `multiset` before `operator=` executes are discarded.

Example

C++

```
// multiset_operator_as.cpp
// compile with: /EHsc
#include <multiset>
#include <iostream>

int main( )
   {
   using namespace std;
   multiset<int> v1, v2, v3;
   multiset<int>::iterator iter;

   v1.insert(10);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
   }
```

`multiset::pointer`

A type that provides a pointer to an element in a `multiset`.

C++

```
typedef typename allocator_type::pointer pointer;
```

Remarks

A type **`pointer`** can be used to modify the value of an element.

In most cases, an iterator should be used to access the elements in a `multiset` object.

`multiset::rbegin`

Returns an iterator addressing the first element in a reversed multiset.

```
C++
```

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed `multiset` or addressing what had been the last element in the unreversed `multiset`.

Remarks

`rbegin` is used with a reversed `multiset` just as rbegin is used with a `multiset`.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, then the `multiset` object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, then the `multiset` object can be modified.

`rbegin` can be used to iterate through a `multiset` backwards.

Example

```
C++
```

```
// multiset_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;

   ms1.insert( 10 );
```

```
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rbegin( );
   cout << "The first element in the reversed multiset is "
        << *ms1_rIter << "." << endl;

   // begin can be used to start an iteration
   // through a multiset in a forward order
   cout << "The multiset is:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration
   // through a multiset in a reverse order
   cout << "The reversed multiset is:";
   for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
      cout << " " << *ms1_rIter;
   cout << endl;

   // a multiset element can be erased by dereferencing to its key
   ms1_rIter = ms1.rbegin( );
   ms1.erase ( *ms1_rIter );

   ms1_rIter = ms1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed multiset is "<< *ms1_rIter << "."
        << endl;
}
```

Output

```
The first element in the reversed multiset is 30.
The multiset is: 10 20 30
The reversed multiset is: 30 20 10
After the erasure, the first element in the reversed multiset is 20.
```

`multiset::reference`

A type that provides a reference to an element stored in a `multiset`.

C++

```
typedef typename allocator_type::reference reference;
```

Example

C++

```
// multiset_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;

   ms1.insert( 10 );
   ms1.insert( 20 );

   // Declare and initialize a reference &Ref1 to the 1st element
   const int &Ref1 = *ms1.begin( );

   cout << "The first element in the multiset is "
        << Ref1 << "." << endl;
}
```

Output

```
The first element in the multiset is 10.
```

`multiset::rend`

Returns an iterator that addresses the location succeeding the last element in a reversed `multiset`.

C++

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed `multiset` (the location that had preceded the first element in the unreversed `multiset`).

Remarks

`rend` is used with a reversed multiset just as end is used with a `multiset`.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the `multiset` object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the `multiset` object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its multiset.

The value returned by `rend` shouldn't be dereferenced.

Example

C++

```
// multiset_rend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;
   multiset <int>::reverse_iterator ms1_rIter;
   multiset <int>::const_reverse_iterator ms1_crIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_rIter = ms1.rend( ) ;
   ms1_rIter--;
   cout << "The last element in the reversed multiset is "
        << *ms1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // through a multiset in a forward order
   cout << "The multiset is: ";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << *ms1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // through a multiset in a reverse order
   cout << "The reversed multiset is: ";
   for ( ms1_rIter = ms1.rbegin( ) ; ms1_rIter != ms1.rend( ); ms1_rIter++ )
      cout << *ms1_rIter << " ";
   cout << "." << endl;
```

```
    ms1_rIter = ms1.rend( );
    ms1_rIter--;
    ms1.erase ( *ms1_rIter );

    ms1_rIter = ms1.rend( );
    --ms1_rIter;
    cout << "After the erasure, the last element in the "
         << "reversed multiset is " << *ms1_rIter << "." << endl;
}
```

`multiset::reverse_iterator`

A type that provides a bidirectional iterator that can read or modify an element in a reversed multiset.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the `multiset` in reverse.

Example

See example for rbegin for an example of how to declare and use `reverse_iterator`.

`multiset::size`

Returns the number of elements in the `multiset`.

C++

```
size_type size() const;
```

Return Value

The current length of the `multiset`.

Example

C++

```
// multiset_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: size_type i;

   ms1.insert( 1 );
   i = ms1.size( );
   cout << "The multiset length is " << i << "." << endl;

   ms1.insert( 2 );
   i = ms1.size( );
   cout << "The multiset length is now " << i << "." << endl;
}
```

Output

```
The multiset length is 1.
The multiset length is now 2.
```

`multiset::size_type`

An unsigned integer type that can represent the number of elements in a `multiset`.

C++

```
typedef typename allocator_type::size_type size_type;
```

Example

See example for size for an example of how to declare and use `size_type`

`multiset::swap`

Exchanges the elements of two multisets.

C++

```
void swap(
    multiset<Key, Compare, Allocator>& right);
```

Parameters

Right

The argument multiset providing the elements to be swapped with the target multiset.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two multisets whose elements are being exchanged.

Example

C++

```
// multiset_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1, ms2, ms3;
   multiset <int>::iterator ms1_Iter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );
   ms2.insert( 100 );
   ms2.insert( 200 );
   ms3.insert( 300 );

   cout << "The original multiset ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;

   // This is the member function version of swap
   ms1.swap( ms2 );

   cout << "After swapping with ms2, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;
```

```
   // This is the specialized template version of swap
   swap( ms1, ms3 );

   cout << "After swapping with ms3, list ms1 is:";
   for ( ms1_Iter = ms1.begin( ); ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original multiset ms1 is: 10 20 30.
After swapping with ms2, list ms1 is: 100 200.
After swapping with ms3, list ms1 is: 300.
```

`multiset::upper_bound`

Returns an iterator to the first element in a `multiset` with a key that is greater than a specified key.

C++

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

`key`
The argument key to be compared with the sort key of an element from the `multiset` being searched.

Return Value

An **iterator** or `const_iterator` that addresses the location of an element in a `multiset` with a key that is greater than the argument key, or that addresses the location succeeding the last element in the `multiset` if no match is found for the key.

Example

C++

```cpp
// multiset_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int> :: const_iterator ms1_AcIter, ms1_RcIter;

   ms1.insert( 10 );
   ms1.insert( 20 );
   ms1.insert( 30 );

   ms1_RcIter = ms1.upper_bound( 20 );
   cout << "The first element of multiset ms1 with a key greater "
        << "than 20 is: " << *ms1_RcIter << "." << endl;

   ms1_RcIter = ms1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( ms1_RcIter == ms1.end( ) )
      cout << "The multiset ms1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of multiset ms1 with a key > 40 is: "
           << *ms1_RcIter << "." << endl;

   // The element at a specific location in the multiset can be
   // found using a dereferenced iterator addressing the location
   ms1_AcIter = ms1.begin( );
   ms1_RcIter = ms1.upper_bound( *ms1_AcIter );
   cout << "The first element of ms1 with a key greater than"
        << endl << "that of the initial element of ms1 is: "
        << *ms1_RcIter << "." << endl;
}
```

Output

```
The first element of multiset ms1 with a key greater than 20 is: 30.
The multiset ms1 doesn't have an element with a key greater than 30.
The first element of ms1 with a key greater than
that of the initial element of ms1 is: 20.
```

`multiset::value_comp`

Retrieves a copy of the comparison object used to order element values in a `multiset`.

C++

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a `multiset` uses to order its elements, which is the template parameter `Compare`.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&** `_xVal`, **const Key&** `_yVal`);

which returns true if `_xVal` precedes and isn't equal to `_yVal` in the sort order.

Both key_compare and value_compare are synonyms for the template parameter `Compare`. Both types are provided for the classes set and `multiset`, where they're identical, for compatibility with the classes map and `multimap`, where they're distinct.

Example

C++

```
// multiset_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   multiset <int, less<int> > ms1;
   multiset <int, less<int> >::value_compare vc1 = ms1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of ms1."
           << endl;
   }
   else
   {
      cout << "vc1( 2,3 ) returns value of false, "
```

```
            << "where vc1 is the function object of ms1."
            << endl;
   }

   set <int, greater<int> > ms2;
   set<int, greater<int> >::value_compare vc2 = ms2.value_comp( );
   bool result2 = vc2( 2, 3 );
   if( result2 == true )
   {
      cout << "vc2( 2,3 ) returns value of true, "
           << "where vc2 is the function object of ms2."
           << endl;
   }
   else
   {
      cout << "vc2( 2,3 ) returns value of false, "
           << "where vc2 is the function object of ms2."
           << endl;
   }
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of ms1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of ms2.
```

`multiset::value_compare`

The type that provides a function object that can compare two sort keys to determine their relative order in the `multiset`.

C++

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Compare`.

Both key_compare and `value_compare` are synonyms for the template parameter `Compare`. Both types are provided for the classes set and `multiset`, where they're identical, for compatibility with the classes map and `multimap`, where they're distinct.

For more information on `Compare`, see the Remarks section of the multiset Class topic.

Example

See the example for value_comp for an example of how to declare and use `value_compare`.

`multiset::value_type`

A type that describes an object stored as an element as a `multiset` in its capacity as a value.

```
C++
```

```
typedef Key value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Key`.

Both key_type and `value_type` are synonyms for the template parameter `Key`. Both types are provided for the classes set and multiset, where they're identical, for compatibility with the classes map and multimap, where they're distinct.

For more information on `Key`, see the Remarks section of the topic.

Example

```
C++
```

```
// multiset_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   multiset <int> ms1;
   multiset <int>::iterator ms1_Iter;

   multiset <int> :: value_type svt_Int;   // Declare value_type
   svt_Int = 10;             // Initialize value_type

   multiset <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type
```

```
   ms1.insert( svt_Int );         // Insert value into s1
   ms1.insert( skt_Int );         // Insert key into s1

   // a multiset accepts key_types or value_types as elements
   cout << "The multiset has elements:";
   for ( ms1_Iter = ms1.begin( ) ; ms1_Iter != ms1.end( ); ms1_Iter++ )
      cout << " " << *ms1_Iter;
   cout << "." << endl;
}
```

Output

```
The multiset has elements: 10 20.
```

See also

Containers

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

<shared_mutex>

Article • 12/07/2021

The `<shared_mutex>` header provides synchronization primitives for protection of shared data that can be accessed by multiple threads. In addition to the exclusive access control provided by mutex classes, the shared mutex classes also allow shared ownership by multiple threads for non-exclusive access. Shared mutexes can be used to control resources that can be read by several threads without causing a race condition, but must be written exclusively by a single thread.

The header `<shared_mutex>` defines the classes `shared_mutex` and `shared_timed_mutex`, the class template `shared_lock`, and the template function `swap` for shared mutex support.

Classes	Description
shared_mutex Class	A shared mutex type that can be locked exclusively by one agent or shared non-exclusively by multiple agents.
shared_timed_mutex Class	A shared timed mutex type that can be locked exclusively by one agent or shared non-exclusively by multiple agents.
shared_lock Class	A class template that wraps a shared mutex to support timed lock operations and non-exclusive sharing by multiple agents.

Functions	Description
swap	Swaps the content of the shared mutex objects referenced by the function parameters.

Syntax

C++

```
namespace std {
    class shared_mutex;
    class shared_timed_mutex;
    template <class Mutex>
class shared_lock;
    template <class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}
```

Remarks

An instance of the class `shared_mutex` is a *shared mutex type*, a type that controls the shared ownership of a mutex within a scope. A shared mutex type meets all the requirements of a mutex type, as well as members to support shared non-exclusive ownership.

A shared mutex type supports the additional methods `lock_shared`, `unlock_shared`, and `try_lock_shared`:

- The `lock_shared` method blocks the calling thread until the thread obtains shared ownership of the mutex.
- The `unlock_shared` method releases shared ownership of the mutex held by the calling thread.
- The `try_lock_shared` method tries to obtain shared ownership of the mutex without blocking. Its return type is convertible to **`bool`** and is **`true`** if the method obtains ownership, but is otherwise **`false`**.

The class `shared_timed_mutex` is a *shared timed mutex type*, a type that meets the requirements of both a shared mutex type and a timed mutex type.

A shared timed mutex type supports the additional methods `try_lock_shared_for` and `try_lock_shared_until`:

- The `try_lock_shared_for` method attempts to obtain shared ownership of the mutex until the duration specified by the parameter has passed. If the duration is not positive, the method is equivalent to `try_lock_shared`. The method does not return within the duration specified unless shared ownership is obtained. Its return value is **`true`** if the method obtains ownership, but is otherwise **`false`**.
- The `try_lock_shared_until` method attempts to obtain shared ownership of the mutex until the specified absolute time has passed. If the specified time has already passed, the method is equivalent to `try_lock_shared`. The method does not return before the time specified unless shared ownership is obtained. Its return value is **`true`** if the method obtains ownership, but is otherwise **`false`**.

The `shared_lock` class template extends support for timed locking and transfer of ownership to a shared mutex. Ownership of the mutex may be obtained at or after construction, and may be transferred to another `shared_lock` object. Objects of type `shared_lock` can be moved, but not copied.

> ⚠ **Warning**
>
> Beginning in Visual Studio 2015, the C++ Standard Library synchronization types are based on Windows synchronization primitives and no longer use ConcRT (except when the target platform is Windows XP). The types defined in `<shared_mutex>` should not be used with any ConcRT types or functions.

Classes

shared_mutex Class

Class `shared_mutex` implements a non-recursive mutex with shared ownership semantics.

C++

```
class shared_mutex {
public:
   shared_mutex();
   ~shared_mutex();
   shared_mutex(const shared_mutex&) = delete;
   shared_mutex& operator=(const shared_mutex&) = delete;
   // Exclusive ownership
   void lock();
   // blocking
   bool try_lock();
   void unlock();
   // Shared ownership
   void lock_shared();
   // blocking
   bool try_lock_shared();
   void unlock_shared();
   // Getters
   typedef void** native_handle_type; // implementation defined
   native_handle_type native_handle();
   };
```

shared_timed_mutex Class

Class `shared_timed_mutex` implements a non-recursive mutex with shared ownership semantics that meets the requirements of a timed mutex type.

C++

```
class shared_timed_mutex {
public:
   shared_timed_mutex();
   ~shared_timed_mutex();
   shared_timed_mutex(const shared_timed_mutex&) = delete;
   shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
   // Exclusive ownership
   void lock();
   // blocking
   bool try_lock();
   template <class Rep, class Period>
   bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
   template <class Clock, class Duration>
   bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
   void unlock();
   // Shared ownership
   void lock_shared();
   // blocking
   bool try_lock_shared();
   template <class Rep, class Period>
   bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
   template <class Clock, class Duration>
   bool try_lock_shared_until(const chrono::time_point<Clock, Duration>&
abs_time);
   void unlock_shared();
   };
```

shared_lock Class

Class template `shared_lock` controls the shared ownership of a shared mutex object within a scope. The template parameter must be a shared mutex type.

C++

```
class shared_lock {
public:
   typedef Mutex mutex_type;
   shared_lock() noexcept;
   explicit shared_lock(mutex_type& m);
   // blocking
   shared_lock(mutex_type& m, defer_lock_t) noexcept;
   shared_lock(mutex_type& m, try_to_lock_t);
   shared_lock(mutex_type& m, adopt_lock_t);
   template <class Clock, class Duration>
   shared_lock(mutex_type& m,
   const chrono::time_point<Clock, Duration>& abs_time);
   template <class Rep, class Period>
   shared_lock(mutex_type& m,
   const chrono::duration<Rep, Period>& rel_time);
   ~shared_lock();
```

```
    shared_lock(shared_lock const&) = delete;
    shared_lock& operator=(shared_lock const&) = delete;
    shared_lock(shared_lock&& u) noexcept;
    shared_lock& operator=(shared_lock&& u) noexcept;
    void lock();
    // blocking
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();
    // Setters
    void swap(shared_lock& u) noexcept;
    mutex_type* release() noexcept;
    // Getters
    bool owns_lock() const noexcept;
    explicit operator bool () const noexcept;
    mutex_type* mutex() const noexcept;
private:
    mutex_type* pm; // exposition only
    bool owns; // exposition only
    };
```

Functions

swap

Swaps the `shared_lock` objects.

C++

```
template <class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
```

Exchanges the content of two `shared_lock` objects. Effectively the same as `x.swap(y)`.

Requirements

Header: <shared_mutex>

Namespace: std

See also

<mutex>

Article • 10/25/2022

A `span` is a view over a contiguous sequence of objects. It provides fast and bounds-safe access. Unlike `vector` or `array`, it doesn't "own" the elements.

See span class for detailed information. Here's an example of how to use a `span`:

C++

```
#include <span>
#include <iostream>

void Show(std::span<int> someValues)
{
    // show values in reverse
    for (auto rIt = someValues.rbegin(); rIt != someValues.rend(); ++rIt)
    {
        std::cout << *rIt;
    }

    // show a subspan
    for (auto& i : someValues.subspan(1, 2))
    {
        std::cout << i;
    }
}

int main()
{
    int numbers[]{ 0,1,2,3,4 };
    Show(numbers); // note conversion from array to span
}
```

Requirements

Header: `<span>`

Namespace: `std`

Compiler option: /std:c++20 or later is required.

Members

Classes

Name	Description
span	Provides a view over a contiguous sequence of objects.

Operators

Name	Description
operator=	Span assignment
operator[]	Element access

Functions

Name	Description
as_bytes	Get the underlying read-only bytes of the span.
as_writable_bytes	Get the underlying bytes of the span.

Constants

Name	Description
`dynamic_extent`	Indicates that the span size is determined at runtime rather than compile time. When the number of elements in the span is known at compile time, it's specified as the `Extent` template parameter. When the number isn't known until runtime, specify `dynamic_extent` instead.

See also

Header Files Reference

`<span>` functions

Article • 06/15/2022

The `<span>` header includes the following non-member functions that operate on `span` objects.

Non-member functions	Description
as_bytes	Get a read-only view of the object representation of the elements in the span.
as_writable_bytes	Get a read/write view of the object representation of the elements in the span.

`as_bytes`

Get a read-only view of the object representation of the elements in the span.

C++

```
template <class T, size_t Extent>
auto as_bytes(span<T, Extent> s) noexcept;
```

Parameters

`T`

The type of the elements in the span.

`Extent`

The number of elements in the span (if known at compile time), otherwise `dynamic_extent` indicating that the number of elements isn't known until runtime.

`s`

The span to get the raw representation of.

Return Value

A `span<const byte, S>` to the first item stored in the span where `S` is `{reinterpret_cast<const std::byte*>(s.data()), s.size_bytes()}`

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

void main()
{
    int a[] = { 0,1,2 };
    span <int> mySpan(a);
    auto bytes = std::as_bytes(mySpan);
}
```

`as_writable_bytes`

If `T` isn't **`const`**, gets a read/write view of the raw byte representation of the elements in the span.

C++

```
template <class T, size_t Extent>
auto as_writable_bytes(span<T, Extent> s) noexcept;
```

Parameters

`T`

The type of the elements in the span.

`Extent`

The number of elements in the span (if known at compile time), otherwise `dynamic_extent` indicating that the number of elements isn't known until runtime.

`s`

The span to get the raw representation of.

Return Value

A `span<byte, S>` to the first item stored in the span where `S` is `{reinterpret_cast<std::byte*>(s.data()), s.size_bytes()}`

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

void main()
{
    int a[] = { 0,1,2 };
    span <int> mySpan(a);
    auto bytes = as_writable_bytes(mySpan);
}
```

See also

`span` class (C++ Standard Library)

Article • 02/07/2023

Provides a lightweight view over a contiguous sequence of objects. A `span` provides a safe way to iterate over and index into objects that are arranged back-to-back in memory. Such as objects stored in a built-in array, `std::array`, or `std::vector`.

If you typically access a sequence of back-to-back objects using a pointer and an index, a `span` is a safer, lightweight alternative.

The size of a `span` can be set at compile time by specifying it as a template argument, or at runtime by specifying `dynamic_extent`.

Syntax

```
C++

template<class T, size_t Extent = dynamic_extent>
class span;
```

Template parameters

`T`
The type of the elements in the `span`.

`Extent`
The number of elements in the `span` if specified at compile time. Otherwise `std::dynamic_extent` if the number of elements will be specified at run-time.

Deduction guide

Members

Type Definitions	Description
const_pointer	The type of a pointer to a `const` element.
const_reference	The type of a reference to a `const` element.

Type Definitions	Description
difference_type	The type of a signed distance between two elements.
element_type	The type of a `span` element.
iterator	The type of an iterator for a `span`.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
reverse_iterator	The type of a reverse iterator for a `span`.
size_type	The type for the result of the unsigned distance between two elements in the `span`.
value_type	The type of an element, without **`const`** or **`volatile`** qualifications.
Constructor	**Description**
span	Construct a `span`.
Iterator support	**Description**
begin	Get an iterator pointing to the first element in the `span`.
end	Get an iterator pointing to the end of the `span`.
rbegin	Get a reverse iterator pointing to the last element of the `span`; that is, the beginning of the reversed `span`.
rend	Get a reverse iterator pointing to the front of the `span`; that is, the end of the reversed `span`.
Access elements	**Description**
back	Get the last element in the `span`.
data	Get the address of the first element in the `span`.
front	Get the first element in the `span`.
operator[]	Access an element at a specified position.
Observers	**Description**
empty	Test whether the `span` is empty.

Type Definitions	Description
size	Get the number of elements in the `span`.
size_bytes	Get the size of the `span` in bytes.
Subviews	**Description**
first	Get a subspan from the front of the `span`.
last	Get a subspan from the back of the `span`.
subspan	Get a subspan from anywhere in the `span`.
Operators	**Description**
span::operator=	Replace the `span`.
span::operator[]	Get the element at the specified position.

Remarks

All `span` member functions have constant time complexity.

Unlike `array` or `vector`, a `span` doesn't "own" the elements inside it. A `span` doesn't free any storage for the items inside it because it doesn't own the storage for those objects.

Requirements

Header: `<span>` (since C++ 20)

Namespace: `std`

Compiler Option: /std:c++20 or later is required.

span::back

Get the last element in the `span`.

C++

```
constexpr reference back() const noexcept;
```

Return value

A reference to the last element in the `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    cout << mySpan.back();
}
```

Output

```
2
```

`span::begin`

Get an iterator pointing at the first element in the `span`.

C++

```
constexpr iterator begin() const noexcept;
```

Return value

An iterator pointing at the first element in the `span`.

Example

C++

```
#include <span>
#include <iostream>
```

```
using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    auto i = mySpan.begin();
    cout << *i;
}
```

Output

```
0
```

`span::data`

Get a pointer to the beginning of the `span` data.

C++

```
constexpr pointer data() const noexcept;
```

Return value

A pointer to the first item stored in the `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    auto i = mySpan.data();
    cout << *i;
}
```

Output

```
0
```

`span::difference_type`

The number of elements between two elements in a `span`.

C++

```
using difference_type = std::ptrdiff_t;
```

Example

C++

```
#include <span>
#include <iostream>

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::difference_type distance = mySpan.end() - mySpan.begin();
    cout << distance << std::endl;
}
```

Output

```
3
```

`span::element_type`

The type of the elements in the `span`.

C++

```
using element_type = T;
```

Remarks

The type is taken from the template parameter `T` when a `span` is created.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::element_type et = mySpan[2];
    cout << et << endl;
}
```

Output

```
2
```

`span::empty`

Whether the `span` contains elements.

C++

```
constexpr bool empty() const noexcept;
```

Return value

Returns **`true`** if `this->size() == 0`. Otherwise **`false`**.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;
```

```
int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    bool isEmpty = mySpan.empty(); // isEmpty == false
}
```

`span::end`

Get an iterator to the end of the `span`.

C++

```
constexpr iterator end() const noexcept;
```

Return value

An iterator pointing just beyond the end of the `span`.

Remarks

`end` is used to test whether an iterator has passed the end of its range.

Don't dereference the value returned by this iterator. Use it to identify whether the iterator has reached beyond the last element in the `span`.

Example

C++

```
// Iteration
for (auto it = s1.begin(); it != s1.end(); ++it)
{
    cout << *it;
}
```

`span::first`

Get a subspan, taken from the front of this `span`.

C++

```
constexpr auto first(size_type count) const noexcept;
template <size_t count> constexpr auto first() const noexcept;
```

Parameters

count

The number of elements from the front of this `span` to put in the subspan.

The number of elements is specified as a parameter to the template, or to the function, as illustrated below.

Return value

A `span` that contains `count` elements from the front of this `span`.

Remarks

Use the template version of this function when possible to validate the `count` at compile time, and to preserve info about the `span` since it returns a `span` of fixed extent.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    auto first2 = mySpan.first(2);
    cout << "mySpan.first(2): ";
    for (auto& i : first2)
    {
        cout << i;
    }

    cout << "\nmySpan.first<2>: ";
    auto viewSpan = mySpan.first<2>();
    for (auto& i : viewSpan)
    {
        cout << i;
```

```
    }
}
```

Output

```
mySpan.first(2): 01
mySpan.first<2>: 01
```

`span::front`

Get the first element in the `span`.

C++

```
constexpr reference front() const noexcept;
```

Return value

A reference to the first element in the `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    auto i = mySpan.front();
    cout << i;
}
```

Output

```
0
```

`span::iterator`

The type of an `iterator` over `span` elements.

C++

```
using iterator = implementation-defined-iterator-type;
```

Remarks

This type serves as an `iterator` over the elements in a `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::iterator it = mySpan.begin();
    cout << *it++ << *it++ << *it;
}
```

Output

```
012
```

`span::last`

Get a subspan, taken from the end of this `span`.

C++

```
constexpr span<element_type, dynamic_extent> last(const size_type count)
const noexcept;
template <size_t count> constexpr span<element_type, count> last() const
noexcept;
```

Parameters

count

The number of elements from the end this `span` to put in the subspan. The number can be specified as a parameter to the template, or to the function, as illustrated below.

Return value

A `span` containing the last `count` elements from this `span`.

Remarks

Use the template version of this function when possible to validate the `count` at compile time, and to preserve info about the `span` since it returns a `span` of fixed extent.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    auto first2 = mySpan.last(2);
    cout << "mySpan.last(2): ";
    for (auto& i : last2)
    {
        cout << i;
    }

    cout << "\nmySpan.last<2>: ";
    auto viewSpan = mySpan.last<2>();
    for (auto& i : viewSpan)
    {
        cout << i;
    }
}
```

Output

```
mySpan.last(2): 12
mySpan.last<2>: 12
```

`span::operator[]`

Get the element in the `span` at a specified position.

C++

```
constexpr reference operator[](size_type offset) const;
```

Parameters

`offset`
Zero-based element in the `span` to access.

Return value

A reference to the element at position *`offset`*. If the position is invalid, the behavior is undefined.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    cout << mySpan[1];
}
```

Output

```
1
```

`span::operator=`

Assign another `span` to this one.

```
C++
```

```
constexpr span& operator=(const span& other) noexcept = default;
```

Parameters

`other`
The `span` to assign to this one.

Return value

`*this`

Remarks

Assignment does a shallow copy of the data pointer and the size. A shallow copy is safe because a `span` doesn't allocate memory for the elements it contains.

Example

```
C++
```

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    span<int> mySpan2;
    mySpan2 = mySpan;
    for (auto &i : mySpan2)
    {
        cout << it;
    }
}
```

Output

```
012
```

span::pointer

The types for a pointer, and `const` pointer, to a `span` element.

C++

```
using pointer = T*;
using const_pointer = const T*;
```

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    // pointer
    span<int>::pointer ptr = &mySpan[2];
    *ptr = 9;
    cout << mySpan[2];

    // const pointer
    span<int>::const_pointer cPtr = &mySpan[0];
    // *cPtr = 9; error - const
    cout << *cPtr;
}
```

Output

```
90
```

span::rbegin

Get a reverse iterator pointing to the last element of this `span`.

C++

```
constexpr reverse_iterator rbegin() const noexcept;
```

Return value

An iterator pointing to the beginning of the reversed `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    for (auto rIt = s1.rbegin(); rIt != s1.rend(); ++rIt)
    {
        cout << *rIt;
    }
}
```

Output

```
210
```

`span::reference`

The types for a reference, and a **`const`** reference, to a `span` element.

C++

```
using reference = T&;
using const_reference = const T&;
```

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    // reference
    span<int>::reference ref = mySpan[0];
    ref = 9;
    cout << mySpan[0];
    // const reference
    span<int>::const_reference cRef = mySpan[1];
    // cRef = 9; error because const
    cout << cRef;
}
```

Output

```
91
```

`span::rend`

Get a random-access iterator that points just beyond the end of the reversed `span`.

C++

```
constexpr reverse_iterator rend() const noexcept;
```

Return value

A reverse iterator to the placeholder following the last element in the reversed `span`; that is, the placeholder before the first element in the unreversed `span`.

Remarks

`rend` is used with a reversed `span` just as span::end is used with a `span`. Use it to test to whether a reverse iterator has reached the end of its `span`.

The value returned by `rend` shouldn't be dereferenced.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    for (auto rIt = s1.rbegin(); rIt != s1.rend(); ++rIt)
    {
        cout << *rIt;
    }
}
```

`span::reverse_iterator`

The type of a reverse iterator for a `span`.

C++

```
using reverse_iterator = std::reverse_iterator<iterator>;
```

Example

C++

```
#include <span>
#include <iostream>

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::reverse_iterator rIt = mySpan.rbegin();
```

```
    cout << *rIt++ << *rIt++ << *rIt;
}
```

Output

```
210
```

`span::size`

Get the number of elements in the `span`.

C++

```
constexpr size_t size() const noexcept;
```

Return value

The number of elements in the `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    cout << mySpan.size();
}
```

Output

```
3
```

`span::size_bytes`

Get the size of the elements in the `span` in bytes.

C++

```
constexpr size_type size_bytes() const noexcept;
```

Return value

The number of bytes that all of the elements in the `span` occupy; that is, `sizeof(element_type)` multiplied by the number of elements in the `span`.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);
    cout << mySpan.size_bytes(); // 3 elements * 4 (size of an int)
}
```

Output

```
12
```

`span::size_type`

An unsigned type, suitable for storing the number of elements in a `span`.

C++

```
using size_type = size_t;
```

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::size_type szType = mySpan.size();
    cout << szType;
}
```

Output

```
3
```

span::span

`span` constructors.

C++

```
constexpr span() noexcept
requires (Extent == 0 || Extent == dynamic_extent) = default;

template <class It>
constexpr explicit(Extent != dynamic_extent)
span(It first, size_type count) noexcept

template <class It, class End>
constexpr explicit(Extent != dynamic_extent)
span(It first, End last) noexcept(noexcept(last - first))

template <class T, size_t N>
requires (Extent == dynamic_extent || Extent == N) && is_convertible_v<T (*)
[], T (*)[]>
constexpr span(array<T, N>& arr) noexcept

template <class T, size_t N>
requires (Extent == dynamic_extent || Extent == N) && is_convertible_v<const
T (*)[], T (*)[]>
constexpr span(const array<T, N>& arr) noexcept

template <size_t N>
requires (Extent == dynamic_extent || Extent == N)
constexpr span(type_identity_t<T> (&arr)[N]) noexcept
```

```
template <class R>
constexpr explicit(Extent != dynamic_extent)
span(R&& r)

// default copy ctor
constexpr span(const span& other) noexcept = default;

// converting  ctor
template <class T, size_t OtherExtent>
requires (Extent == dynamic_extent || OtherExtent == dynamic_extent ||
              Extent == OtherExtent) && is_convertible_v<T (*)[], T (*)[]>
constexpr explicit(Extent != dynamic_extent && OtherExtent ==
dynamic_extent)
span(const span<T, OtherExtent>& other) noexcept
```

Parameters

arr

Construct a `span` from an array.

count

Number of elements that will be in the `span`.

first

Iterator to the first element in the `span`.

last

Iterator to just past the last element in the `span`.

N

The number of elements that will be in the `span`.

other

Make a copy of this `span`.

r

Construct a `span` from the range `R`.

Remarks

A `span` doesn't free storage for items in the `span` because it doesn't own the storage of the objects within it.

Constructor	Description

Constructor	Description
`span()`	Construct an empty `span`. Only considered during overload resolution when the template parameter `Extent` is `0` or `dynamic_extent`.
`span(It first, size_type count)`	Construct a `span` from the first `count` elements from iterator `first`. Only considered during overload resolution when template parameter `Extent` isn't `dynamic_extent`.
`span(It first, End last)`	Construct a `span` from the elements in iterator `first` until the end `last` is reached. Only considered during overload resolution when template parameter `Extent` isn't `dynamic_extent`. `It` must be a contiguous_iterator.
`span(array<T, N>& arr) noexcept;` `span(const array<T, N>& arr) noexcept;` `span(type_identity_t<element_type> (&arr)[N]) noexcept;`	Construct a `span` from `N` elements of the specified array. Only considered during overload resolution when template parameter `Extent` is `dynamic_extent` or equals `N`.
`span(R&& r)`	Construct a `span` from a range. Only participates in overload resolution if template parameter `Extent` isn't `dynamic_extent`.
`span(const span& other)`	The compiler-generated copy constructor. A shallow copy of the data pointer is safe because the `span` doesn't allocate the memory to hold the elements.
`span(const span<OtherElementType, OtherExtent>& s) noexcept;`	Converting constructor: construct a `span` from another `span`. Only participates in overload resolution if template parameter `Extent` is `dynamic_extent`, or `N` is `dynamic_extent` or equals `Extent`.

Example

C++

```
#include <span>

using namespace std;

int main()
{
    const int MAX=10;

    int x[MAX];
```

```
    for (int i = 0; i < MAX; i++)
    {
        x[i] = i;
    }

    span<int, MAX> span1{ x }; // fixed-size span: compiler error if size of
x doesn't match template argument MAX
    span<int> span2{ x }; // size is inferred from x
    span<const int> span3 = span2; // converting constructor
    span<int> span4( span2 ); // copy constructor
}
```

span::subspan

Get a subspan of this `span`.

C++

```
constexpr auto subspan(size_type offset, size_type count = dynamic_extent)
const noexcept;

template <size_t offset, size_t count = dynamic_extent>
constexpr auto subspan() const noexcept
```

Parameters

`count`

The number of elements to put in the subspan. If `count` is `dynamic_extent` (the default value), then the subspan is taken from `offset` to the end of this `span`.

`offset`

The location in this `span` to start the subspan.

Return value

A `span` starting at `offset` in this `span`. Contains `count` elements.

Remarks

A template version of this function is available that checks the count at compile time, which preserves information about the `span` by returning a `span` of fixed extent.

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    cout << "mySpan.subspan(1,2): ";
    for (auto& i : mySpan.subspan(1,2))
    {
        cout << i;
    }
    cout << "\nmySpan.subspan<1,2>: ";
    for (auto& i : mySpan.subspan<1,2>())
    {
        cout << i;
    }
    cout << "\nmySpan.subspan<1>: ";
    for (auto& i : mySpan.subspan<1>)
    {
        cout << i;
    }
}
```

Output

```
mySpan.subspan(1,2): 12
mySpan.subspan<1,2>: 12
mySpan.subspan<1>: 12
```

`span::value_type`

The type of the element in the `span`, without **`const`** or **`volatile`** qualifications.

C++

```
using value_type = std::remove_cv_t<T>;
```

Example

C++

```
#include <span>
#include <iostream>

using namespace std;

int main()
{
    int a[] = { 0,1,2 };
    span<int> mySpan(a);

    span<int>::value_type vType = mySpan[2];
    cout << vType;
}
```

Output

```
2
```

Deduction guides

The following deduction guides are provided for `span`.

C++

```
// Allows the extent to be deduced from std::array and C++ built-in arrays

template <class T, size_t Extent>
span(T (&)[Extent]) -> span<T, Extent>;

template <class T, size_t Size>
span(array<T, Size>&) -> span<T, Size>;

template <class T, size_t Size>
span(const array<T, Size>&) -> span<const T, Size>;

// Allows the element type to be deduced from the iterator and the end of
the span.
// The iterator must be contiguous

template <contiguous_iterator It, class End>
span(It, End) -> span<remove_reference_t<iter_reference_t<It>>>;

// Allows the element type to be deduced from a range.
// The range must be contiguous

template <ranges::contiguous_range Rng>
span(Rng &&) -> span<remove_reference_t<ranges::range_reference_t<Rng>>>;
```

See also

How to use class template argument deduction

<sstream>

Article • 12/07/2021

Defines several class templates that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of class template basic_string.

Syntax

C++

```
namespace std {
template <class CharType, class Traits = char_traits<CharType>, class
Allocator = allocator<CharType>>
class basic_stringbuf;
typedef basic_stringbuf<char>
stringbuf;
typedef basic_stringbuf<wchar_t> wstringbuf;

template <class CharType, class Traits = char_traits<CharType>, class
Allocator = allocator<CharType>>
class basic_istringstream;
typedef basic_istringstream<char>
istringstream;
typedef basic_istringstream<wchar_t> wistringstream;

template <class CharType, class Traits = char_traits<CharType>, class
Allocator = allocator<CharType>>
class basic_ostringstream;
typedef basic_ostringstream<char>
ostringstream;
typedef basic_ostringstream<wchar_t> wostringstream;

template <class CharType, class Traits = char_traits<CharType>, class
Allocator = allocator<CharType>>
class basic_stringstream;
typedef basic_stringstream<char>
stringstream;
typedef basic_stringstream<wchar_t> wstringstream;
// TEMPLATE FUNCTIONS
template <class CharType, class Traits, class Allocator>
void swap(
    basic_stringbuf<CharType, Traits, Allocator>& left,
    basic_stringbuf<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
void swap(
    basic_istringstream<CharType, Traits, Allocator>& left,
    basic_istringstream<CharType, Traits, Allocator>& right);
```

```
template <class CharType, class Traits, class Allocator>
void swap(
    basic_ostringstream<CharType, Traits, Allocator>& left,
    basic_ostringstream<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
void swap (
    basic_stringstream<CharType, Traits, Allocator>& left,
    basic_stringstream<CharType, Traits, Allocator>& right);

}  // namespace std
```

Parameters

left
Reference to an `sstream` object.

right
Reference to an `sstream` object.

Remarks

Objects of type `char *` can use the functionality in <strstream> for streaming. However, <strstream> is deprecated and the use of <sstream> is encouraged.

Typedefs

Type name	Description
istringstream	Creates a type `basic_istringstream` specialized on a `char` template parameter.
ostringstream	Creates a type `basic_ostringstream` specialized on a `char` template parameter.
stringbuf	Creates a type `basic_stringbuf` specialized on a `char` template parameter.
stringstream	Creates a type `basic_stringstream` specialized on a `char` template parameter.
wistringstream	Creates a type `basic_istringstream` specialized on a `wchar_t` template parameter.
wostringstream	Creates a type `basic_ostringstream` specialized on a `wchar_t` template parameter.
wstringbuf	Creates a type `basic_stringbuf` specialized on a `wchar_t` template parameter.

Type name	Description
wstringstream	Creates a type `basic_stringstream` specialized on a `wchar_t` template parameter.

Manipulators

Name	Description
swap	Exchanges the values between two `sstream` objects.

Classes

Class	Description
basic_stringbuf	Describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are determined by the class `Tr`, to and from a sequence of elements stored in an array object.
basic_istringstream	Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf<**Elem**, **Tr**, `Alloc`>, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`.
basic_ostringstream	Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_stringbuf<**Elem**, **Tr**, `Alloc`>, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`.
basic_stringstream	Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf<**Elem**, **Tr**, `Alloc`>, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`.

Requirements

- **Header:** <sstream>
- **Namespace:** std

See also

Header Files Reference

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

`<sstream>` functions

Article • 12/07/2021

swap

swap

Exchanges the values between two `sstream` objects.

C++

```
template <class Elem, class Tr, class Alloc>
void swap(
    basic_stringbuf<Elem, Tr, Alloc>& left,
    basic_stringbuf<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_istringstream<Elem, Tr, Alloc>& left,
    basic_istringstream<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_ostringstream<Elem, Tr, Alloc>& left,
    basic_ostringstream<Elem, Tr, Alloc>& right);

template <class Elem, class Tr, class Alloc>
void swap(
    basic_stringstream<Elem, Tr, Alloc>& left,
    basic_stringstream<Elem, Tr, Alloc>& right);
```

Parameters

left

Reference to an `sstream` object.

right

Reference to an `sstream` object.

Remarks

The template function executes `left.swap(right)`.

See also

<sstream>

`<sstream>` typedefs

Article • 03/02/2022

istringstream
ostringstream
stringbuf
stringstream
wistringstream
wostringstream
wstringbuf
wstringstream

`istringstream`

Creates a type `basic_istringstream` specialized on a `char` template parameter.

```
C++
```

```
typedef basic_istringstream<char> istringstream;
```

Remarks

The type is a synonym for class template basic_istringstream, specialized for elements of type `char`.

`ostringstream`

Creates a type `basic_ostringstream` specialized on a `char` template parameter.

```
C++
```

```
typedef basic_ostringstream<char> ostringstream;
```

Remarks

The type is a synonym for class template basic_ostringstream, specialized for elements of type `char`.

`stringbuf`

Creates a type `basic_stringbuf` specialized on a `char` template parameter.

```
C++
```

```
typedef basic_stringbuf<char> stringbuf;
```

Remarks

The type is a synonym for class template basic_stringbuf, specialized for elements of type `char`.

`stringstream`

Creates a type `basic_stringstream` specialized on a `char` template parameter.

```
C++
```

```
typedef basic_stringstream<char> stringstream;
```

Remarks

The type is a synonym for class template basic_stringstream, specialized for elements of type `char`.

`wistringstream`

Creates a type `basic_istringstream` specialized on a `wchar_t` template parameter.

```
C++
```

```
typedef basic_istringstream<wchar_t> wistringstream;
```

Remarks

The type is a synonym for class template basic_istringstream, specialized for elements of type `wchar_t`.

`wostringstream`

Creates a type `basic_ostringstream` specialized on a `wchar_t` template parameter.

```
C++

typedef basic_ostringstream<wchar_t> wostringstream;
```

Remarks

The type is a synonym for class template basic_ostringstream, specialized for elements of type `wchar_t`.

`wstringbuf`

Creates a type `basic_stringbuf` specialized on a `wchar_t` template parameter.

```
C++

typedef basic_stringbuf<wchar_t> wstringbuf;
```

Remarks

The type is a synonym for class template basic_stringbuf, specialized for elements of type `wchar_t`.

`wstringstream`

Creates a type `basic_stringstream` specialized on a `wchar_t` template parameter.

```
C++

typedef basic_stringstream<wchar_t> wstringstream;
```

Remarks

The type is a synonym for class template basic_stringstream, specialized for elements of type `wchar_t`.

See also

<sstream>

basic_stringbuf Class

Article • 06/16/2022

Describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are determined by the class `Tr`, to and from a sequence of elements stored in an array object.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>,
    class Alloc = allocator<Elem>>
class basic_stringbuf : public basic_streambuf<Elem, Tr>
```

Parameters

Alloc
The allocator class.

Elem
The type of the basic element of the string.

Tr
The character traits specialized on the basic element of the string.

Remarks

The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class basic_stringbuf< `Elem`, `Tr`, `Alloc` > stores a copy of the `ios_base::`openmode argument from its constructor as its `stringbuf` mode **mode**:

- If `mode & ios_base::in` is nonzero, the input buffer is accessible. For more information, see basic_streambuf Class.
- If `mode & ios_base::out` is nonzero, the output buffer is accessible.

Constructors

Constructor	Description
basic_stringbuf	Constructs an object of type `basic_stringbuf`.

Typedefs

Type name	Description
allocator_type	The type is a synonym for the template parameter *Alloc*.
char_type	Associates a type name with the *Elem* template parameter.
int_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the *Tr* scope.
off_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the *Tr* scope.
pos_type	Makes this type within `basic_filebuf`'s scope equivalent to the type of the same name in the *Tr* scope.
traits_type	Associates a type name with the *Tr* template parameter.

Member functions

Member function	Description
overflow	A protected, virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	The protected virtual member function tries to put back an element into the input buffer, then makes it the current element (pointed to by the next pointer).
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
str	Sets or gets the text in a string buffer without changing the write position.
swap	
underflow	The protected virtual member function to extract the current element from the input stream.

Requirements

Header: <sstream>

Namespace: std

basic_stringbuf::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
C++
```

```
typedef Alloc allocator_type;
```

basic_stringbuf::basic_stringbuf

Constructs an object of type `basic_stringbuf`.

```
C++
```

```
basic_stringbuf(
    ios_base::openmode _Mode = ios_base::in | ios_base::out);

basic_stringbuf(
    const basic_string<Elem, Tr, Alloc>& str,
    ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

_Mode
One of the enumerations in ios_base::openmode.

str
An object of type basic_string.

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. For more information, see the Remarks section of the basic_streambuf Class. It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the basic_stringbuf Class.

The second constructor allocates a copy of the sequence controlled by the string object *str*. If `_Mode & ios_base::in` is nonzero, it sets the input buffer to start reading at the start of the sequence. If `_Mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores *_Mode* as the stringbuf mode. For more information, see the Remarks section of the basic_stringbuf Class.

basic_stringbuf::char_type

Associates a type name with the *Elem* template parameter.

```
C++
```

```
typedef Elem char_type;
```

basic_stringbuf::int_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the `Tr` scope.

```
C++
```

```
typedef typename traits_type::int_type int_type;
```

basic_stringbuf::off_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the `Tr` scope.

```
C++
```

```
typedef typename traits_type::off_type off_type;
```

basic_stringbuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
C++
```

```
virtual int_type overflow(int_type _Meta = traits_type::eof());
```

Parameters

_Meta
The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function can't succeed, it returns `traits_type::eof`. Otherwise, it returns **traits_type::**not_eof(_ *Meta*).

Remarks

If *_Meta* doesn't compare equal to **traits_type::**eof, the protected virtual member function tries to insert the element **traits_type::**to_char_type(*_Meta*) into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. Extending the output buffer this way also extends any associated input buffer.

basic_stringbuf::pbackfail

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

```
C++

virtual int_type pbackfail(int_type _Meta = traits_type::eof());
```

Parameters

_Meta
The character to insert into the buffer, or `traits_type::eof`.

Return Value

If the function can't succeed, it returns `traits_type::eof`. Otherwise, it returns **traits_type::**not_eof(_ *Meta*).

Remarks

If _*Meta* compares equal to **traits_type::**eof, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **byte** = **traits_type::**to_char_type(_ *Meta*). The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to byte, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode permits the sequence to be altered (**mode & ios_base::out** is nonzero), it can store byte into the putback position and decrement the next pointer for the input buffer.

basic_stringbuf::pos_type

Makes this type within basic_filebuf's scope equivalent to the type of the same name in the `Tr` scope.

```
C++
```

```
typedef typename traits_type::pos_type pos_type;
```

basic_stringbuf::seekoff

The protected virtual member function tries to alter the current positions for the controlled streams.

```
C++
```

```
virtual pos_type seekoff(
    off_type _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

_Off
The position to seek for relative to *_Way*. For more information, see basic_stringbuf::off_type.

_Way
The starting point for offset operations. See ios_base::seekdir for possible values.

_Mode
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions. For more information, see ios_base::openmode.

Return Value

Returns the new position or an invalid stream position.

Remarks

For an object of class `basic_stringbuf<Elem, Tr, Alloc>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `_Way` == `ios_base::beg`, the new position is the beginning of the stream plus *_Off*.
- If `_Way` == `ios_base::cur`, the new position is the current stream position plus *_Off*.
- If `_Way` == `ios_base::end`, the new position is the end of the stream plus *_Off*.

If `_Mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `_Mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, *_Way* must be `ios_base::beg` or `ios_base::end` and both streams are positioned at the same element. Otherwise (or if neither position is affected), the positioning operation fails.

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::seekpos

The protected virtual member function tries to alter the current positions for the controlled streams.

```
C++
```

```
virtual pos_type seekpos(pos_type _Sp, ios_base::openmode _Mode =
ios_base::in | ios_base::out);
```

Parameters

_Sp
The position to seek for.

_Mode
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both of the stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

For an object of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by _ *Sp*.

If **mode & ios_base::in** is nonzero, the function alters the next position to read in the input buffer. If **mode & ios_base::out** is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected), the positioning operation fails.

basic_stringbuf::str

Sets or gets the text in a string buffer without changing the write position.

```
C++
```

```
basic_string<Elem, Tr, Alloc> str() const;
void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr
The new string.

Return Value

Returns an object of class basic_string< **Elem**, **Tr**, Alloc >, whose controlled sequence is a copy of the sequence controlled by `*this`.

Remarks

The first member function returns an object of class basic_string< **Elem**, **Tr**, `Alloc` >, whose controlled sequence is a copy of the sequence controlled by `*this`. The sequence copied depends on the stored stringbuf mode:

- If **mode & ios_base::out** is nonzero and an output buffer exists, the sequence is the entire output buffer (epptr - pbase elements beginning with `pbase`).
- If **mode & ios_base::in** is nonzero and an input buffer exists, the sequence is the entire input buffer (egptr - eback elements beginning with `eback`).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by `*this`. It then allocates a copy of the sequence controlled by *_Newstr*. If **mode & ios_base::in** is nonzero, it sets the input buffer to start reading at the beginning of the sequence. If **mode & ios_base::out** is nonzero, it sets the output buffer to start writing at the beginning of the sequence.

Example

```
C++
```

```cpp
// basic_stringbuf_str.cpp
// compile with: /EHsc
#include <iostream>
#include <sstream>

using namespace std;

int main( )
{
   basic_string<char> i( "test" );
   stringstream ss;

   ss.rdbuf( )->str( i );
   cout << ss.str( ) << endl;

   ss << "z";
   cout << ss.str( ) << endl;

   ss.rdbuf( )->str( "be" );
   cout << ss.str( ) << endl;
}
```

Output

```
test
zest
be
```

basic_stringbuf::traits_type

Associates a type name with the *Tr* template parameter.

C++

```cpp
typedef Tr traits_type;
```

Remarks

The type is a synonym for the template parameter *Tr*.

basic_stringbuf::underflow

Protected, virtual function to extract the current element from the input stream.

C++

```
virtual int_type underflow();
```

Return Value

If the function can't succeed, it returns **traits_type::**eof. Otherwise, it returns the current element in the input stream, which is converted.

Remarks

The protected virtual member function tries to extract the current element `byte` from the input buffer, advance the current stream position, and return the element as **traits_type::**to_int_type(**byte**). It can do so in one way: If a read position is available, it takes `byte` as the element stored in the read position and advances the next pointer for the input buffer.

basic_streambuf::swap

Swaps the contents of this string buffer with another string buffer.

C++

```
void basic_stringbuf<T>::swap(basic_stringbuf& other)
```

Parameters

other
The basic_stringbuf whose contents will be swapped with this basic_stringbuf.

Remarks

basic_stringbuf::operator=

Assigns the contents of the basic_stringbuf on the right side of the operator to the basic_stringbuf on the left side.

C++

```
basic_stringbuf& basic_stringbuf:: operator=(const basic_stringbuf& other)
```

Parameters

other
A basic_stringbuf whose contents, including locale traits, will be assigned to the stringbuf on the left side of the operator.

Remarks

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

basic_istringstream Class

Article • 06/16/2022

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc =
allocator<Elem>>
class basic_istringstream : public basic_istream<Elem, Tr>
```

Parameters

Alloc
The allocator class.

Elem
The type of the basic element of the string.

Tr
The character traits specialized on the basic element of the string.

Remarks

The class template describes an object that controls extraction of elements and encoded objects from a stream buffer of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >, with elements of type *Elem*, whose character traits are determined by the class *Tr*, and whose elements are allocated by an allocator of class *Alloc*. The object stores an object of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Constructors

Constructor	Description
basic_istringstream	Constructs an object of type `basic_istringstream`.

Typedefs

Type name	Description
allocator_type	The type is a synonym for the template parameter `Alloc`.

Member functions

Member function	Description
rdbuf	Returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< `Elem`, `Tr`, `Alloc` >.
str	Sets or gets the text in a string buffer without changing the write position.
swap	Exchanges the values in this `basic_istringstream` object for the provided object.

Operators

Operator	Description
operator=	Assigns the values to this `basic_istringstream` object from the object parameter.

Requirements

Header: <sstream>

Namespace: std

basic_istringstream::allocator_type

The type is a synonym for the template parameter `Alloc`.

```
C++
```

```
typedef Alloc allocator_type;
```

basic_istringstream::basic_istringstream

Constructs an object of type `basic_istringstream`.

C++

```
explicit basic_istringstream(
    ios_base::openmode _Mode = ios_base::in);

explicit basic_istringstream(
    const basic_string<Elem, Tr, Alloc>& str,
    ios_base::openmode _Mode = ios_base::in);

basic_istringstream(
    basic_istringstream&& right);
```

Parameters

_Mode
One of the enumerations in ios_base::openmode.

str
An object of type `basic_string`.

right
An rvalue reference of a `basic_istringstream` object.

Remarks

The first constructor initializes the base class by calling `basic_istream]( sb )`, where `sb` is the stored object of class `basic_stringbuf< Elem, Tr, Alloc>`. It also initializes `sb` by calling `basic_stringbuf< Elem, Tr, Alloc >( _Mode | ios_base::in )`. For more information, see basic_istream and basic_stringbuf.

The second constructor initializes the base class by calling `basic_istream( sb )`. It also initializes `sb` by calling `basic_stringbuf< Elem, Tr, Alloc >( str, _Mode | ios_base::in )`.

The third constructor initializes the object with the contents of *right*, treated as an rvalue reference.

basic_istringstream::operator=

Assigns the values to this `basic_istringstream` object from the object parameter.

C++

```
basic_istringstream& operator=(basic_istringstream&& right);
```

Parameters

right
An rvalue reference to a `basic_istringstream` object.

Remarks

The member operator replaces the contents of the object with the contents of *right*, treated as an rvalue reference move assignment.

basic_istringstream::rdbuf

Returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

```
C++
```

```
basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Example

See basic_filebuf::close for an example that uses `rdbuf`.

basic_istringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
C++
```

```
basic_string<Elem, Tr, Alloc> str() const;
```

```
void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr
The new string.

Return Value

Returns an object of class basic_string< **Elem**, **Tr**, `Alloc` >, whose controlled sequence is a copy of the sequence controlled by `*this`.

Remarks

The first member function returns rdbuf -> str. The second member function calls `rdbuf` -> **str**(`_Newstr`).

Example

See basic_stringbuf::str for an example that uses `str`.

basic_istringstream::swap

Exchanges the values of two `basic_istringstream` objects.

```
C++
```

```
void swap(basic_istringstream& right);
```

Parameters

right
An lvalue reference to a `basic_istringstream` object.

Remarks

The member function exchanges the values of this object and the values of *right*.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

basic_ostringstream Class

Article • 06/16/2022

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc =
allocator<Elem>>
class basic_ostringstream : public basic_ostream<Elem, Tr>
```

Parameters

Alloc
The allocator class.

Elem
The type of the basic element of the string.

Tr
The character traits specialized on the basic element of the string.

Remarks

The class describes an object that controls insertion of elements and encoded objects into a stream buffer, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`. The object stores an object of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Constructors

Constructor	Description
basic_ostringstream	Constructs an object of type `basic_ostringstream`.

Typedefs

Type name	Description
allocator_type	The type is a synonym for the template parameter *Alloc*.

Member functions

Member function	Description
rdbuf	Returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< `Elem`, `Tr`, `Alloc` >.
str	Sets or gets the text in a string buffer without changing the write position.

Requirements

Header: <sstream>

Namespace: std

basic_ostringstream::allocator_type

The type is a synonym for the template parameter *Alloc*.

```
C++

typedef Alloc allocator_type;
```

basic_ostringstream::basic_ostringstream

Constructs an object of type basic_ostringstream.

```
C++

explicit basic_ostringstream(ios_base::openmode _Mode = ios_base::out);

explicit basic_ostringstream(const basic_string<Elem, Tr, Alloc>& str,
ios_base::openmode _Mode = ios_base::out);
```

Parameters

_Mode
One of the enumerations in ios_base::openmode.

str
An object of type `basic_string`.

Remarks

The first constructor initializes the base class by calling `basic_ostream( sb )`, where `sb` is the stored object of class `basic_stringbuf< Elem, Tr, Alloc>`. It also initializes `sb` by calling `basic_stringbuf< Elem, Tr, Alloc>( _Mode | ios_base::out )`. For more information, see basic_ostream and basic_stringbuf.

The second constructor initializes the base class by calling `basic_ostream( sb )`. It also initializes `sb` by calling `basic_stringbuf< Elem, Tr, Alloc >( str, _Mode | ios_base::out )`.

basic_ostringstream::rdbuf

Returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

```
C++

basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer, of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Example

See basic_filebuf::close for an example that uses `rdbuf`.

basic_ostringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
C++
```

```
basic_string<Elem, Tr, Alloc> str() const;

void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr
The new string.

Return Value

Returns an object of class basic_string< **Elem**, **Tr**, `Alloc` >, whose controlled sequence is a copy of the sequence controlled by `*this`.

Remarks

The first member function returns rdbuf -> str. The second member function calls `rdbuf` -> **str**(`_Newstr`).

Example

See basic_stringbuf::str for an example that uses `str`.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

basic_stringstream Class

Article • 06/16/2022

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Syntax

C++

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc =
allocator<Elem>>
class basic_stringstream : public basic_iostream<Elem, Tr>
```

Parameters

Alloc
The allocator class.

Elem
The type of the basic element of the string.

Tr
The character traits specialized on the basic element of the string.

Remarks

The class template describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >, with elements of type `Elem`, whose character traits are determined by the class `Tr`, and whose elements are allocated by an allocator of class `Alloc`. The object stores an object of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Constructors

Constructor	Description
basic_stringstream	Constructs an object of type `basic_stringstream`.

Typedefs

Type name	Description
allocator_type	The type is a synonym for the template parameter `Alloc`.

Member functions

Member function	Description
rdbuf	Returns the address of the stored stream buffer of type `pointer` to basic_stringbuf< `Elem`, `Tr`, `Alloc` >.
str	Sets or gets the text in a string buffer without changing the write position.

Requirements

Header: <sstream>

Namespace: std

basic_stringstream::allocator_type

The type is a synonym for the template parameter `Alloc`.

```
C++

typedef Alloc allocator_type;
```

basic_stringstream::basic_stringstream

Constructs an object of type `basic_stringstream`.

```
C++

explicit basic_stringstream(ios_base::openmode _Mode = ios_base::in |
ios_base::out);

explicit basic_stringstream(const basic_string<Elem, Tr, Alloc>& str,
ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

_Mode
One of the enumerations in ios_base::openmode.

str
An object of type `basic_string`.

Remarks

The first constructor initializes the base class by calling basic_iostream(**sb**), where `sb` is the stored object of class basic_stringbuf< **Elem**, **Tr**, `Alloc` >. It also initializes `sb` by calling basic_stringbuf< **Elem**, **Tr**, `Alloc` >(`_Mode`).

The second constructor initializes the base class by calling basic_iostream(**sb**). It also initializes `sb` by calling basic_stringbuf< **Elem**, **Tr**, `Alloc` >(_ *Str*, `_Mode`).

basic_stringstream::rdbuf

Returns the address of the stored stream buffer of type **pointer** to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

```
C++

basic_stringbuf<Elem, Tr, Alloc> *rdbuf() const;
```

Return Value

The address of the stored stream buffer of type `pointer` to basic_stringbuf< **Elem**, **Tr**, `Alloc` >.

Example

See basic_filebuf::close for an example that uses `rdbuf`.

basic_stringstream::str

Sets or gets the text in a string buffer without changing the write position.

```
C++
```

```
basic_string<Elem, Tr, Alloc> str() const;

void str(
    const basic_string<Elem, Tr, Alloc>& _Newstr);
```

Parameters

_Newstr
The new string.

Return Value

Returns an object of class basic_string< **Elem**, **Tr**, `Alloc` >, whose controlled sequence is a copy of the sequence controlled by `*this`.

Remarks

The first member function returns rdbuf -> str. The second member function calls `rdbuf` -> **str**(`_Newstr`).

Example

See basic_stringbuf::str for an example that uses `str`.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

<stack>

Article • 12/07/2021

Defines the class template stack and two supporting templates.

Requirements

Header: <stack>

Namespace: std

> ⓘ Note
>
> The <stack> library also uses the `#include <initializer_list>` statement.

Members

Operators

Name	Description
operator!=	Tests if the stack object on the left side of the operator is not equal to the stack object on the right side.
operator<	Tests if the stack object on the left side of the operator is less than the stack object on the right side.
operator<=	Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.
operator==	Tests if the stack object on the left side of the operator is equal to the stack object on the right side.
operator>	Tests if the stack object on the left side of the operator is greater than the stack object on the right side.
operator>=	Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.

Classes

Name	Description
stack Class	A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<stack>` operators

Article • 12/07/2021

operator!=

Tests if the stack object on the left side of the operator is not equal to stack object on the right side.

```
C++
```

```
bool operator!=(const stack <Type, Container>& left, const stack <Type,
Container>& right,);
```

Parameters

left
An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stacks or stacks are not equal; **`false`** if stacks or stacks are equal.

Remarks

The comparison between stacks objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

```
C++
```

```
// stack_op_me.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   // Declares stacks with vector base containers
   stack <int, vector<int> > s1, s2, s3;

   // The following would have cause an error because stacks with
   // different base containers are not equality comparable
   // stack <int, list<int> >  s3;

   s1.push( 1 );
   s2.push( 2 );
   s3.push( 1 );

   if ( s1 != s2 )
      cout << "The stacks s1 and s2 are not equal." << endl;
   else
      cout << "The stacks s1 and s2 are equal." << endl;

   if ( s1 != s3 )
      cout << "The stacks s1 and s3 are not equal." << endl;
   else
      cout << "The stacks s1 and s3 are equal." << endl;
}
```

Output

```
The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.
```

operator<

Tests if the stack object on the left side of the operator is less than the stack object on the right side.

C++

```
bool operator<(const stack <Type, Container>& left, const stack <Type,
Container>& right);
```

Parameters

left

An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stack on the left side of the operator is less than and not equal to the stack on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// stack_op_lt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 2 );
   s1.push( 4 );
   s1.push( 6 );
   s1.push( 8 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 2 );
   s3.push( 4 );
   s3.push( 6 );
   s3.push( 8 );

   if ( s1 >= s2 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s2." << endl;
```

```
    if ( s1>= s3 )
       cout << "The stack s1 is greater than or equal to "
            << "the stack s3." << endl;
    else
       cout << "The stack s1 is less than "
            << "the stack s3." << endl;

    // to print out the stack s1 ( by unstacking the elements):
    stack <int>::size_type i_size_s1 = s1.size( );
    cout << "The stack s1 from the top down is: ( ";
    unsigned int i;
    for ( i = 1 ; i <= i_size_s1 ; i++ )
    {
       cout << s1.top( ) << " ";
       s1.pop( );
    }
    cout << ")." << endl;
}
```

Output

```
The stack s1 is less than the stack s2.
The stack s1 is greater than or equal to the stack s3.
The stack s1 from the top down is: ( 8 6 4 2 ).
```

`operator<=`

Tests if the stack object on the left side of the operator is less than or equal to the stack object on the right side.

C++

```
bool operator<=(const stack <Type, Container>& left, const stack <Type,
Container>& right);
```

Parameters

left
An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stack on the left side of the operator is less than or equal to the stack on the right side of the operator; otherwise `false`.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The less than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// stack_op_le.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with default deque base container
   stack <int> s1, s2, s3;

   s1.push( 5 );
   s1.push( 10 );
   s2.push( 1 );
   s2.push( 2 );
   s3.push( 5 );
   s3.push( 10 );

   if ( s1 <= s2 )
      cout << "The stack s1 is less than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is greater than "
           << "the stack s2." << endl;

   if ( s1 <= s3 )
      cout << "The stack s1 is less than or equal to "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is greater than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is greater than the stack s2.
The stack s1 is less than or equal to the stack s3.
```

operator==

Tests if the stack object on the left side of the operator is equal to stack object on the right side.

C++

```
bool operator==(const stack <Type, Container>& left, const stack <Type,
Container>& right);
```

Parameters

left
An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stacks or stacks are equal; **`false`** if stacks or stacks are not equal.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. Two stacks are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// stack_op_eq.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   // Declares stacks with vector base containers
   stack <int, vector<int> > s1, s2, s3;

   // The following would have cause an error because stacks with
   // different base containers are not equality comparable
   // stack <int, list<int> > s3;

   s1.push( 1 );
   s2.push( 2 );
   s3.push( 1 );

   if ( s1 == s2 )
      cout << "The stacks s1 and s2 are equal." << endl;
   else
      cout << "The stacks s1 and s2 are not equal." << endl;

   if ( s1 == s3 )
      cout << "The stacks s1 and s3 are equal." << endl;
   else
      cout << "The stacks s1 and s3 are not equal." << endl;
}
```

Output

```
The stacks s1 and s2 are not equal.
The stacks s1 and s3 are equal.
```

`operator>`

Tests if the stack object on the left side of the operator is greater than the stack object on the right side.

C++

```
bool operator>(const stack <Type, Container>& left, const stack <Type,
Container>& right);
```

Parameters

left

An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stack on the left side of the operator is greater than and not equal to the stack on the right side of the operator; otherwise **`false`**.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater-than relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

C++

```
// stack_op_gt.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 1 );
   s1.push( 2 );
   s1.push( 3 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 1 );
   s3.push( 2 );

   if ( s1 > s2 )
      cout << "The stack s1 is greater than "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is not greater than "
           << "the stack s2." << endl;

   if ( s1> s3 )
      cout << "The stack s1 is greater than "
```

```
            << "the stack s3." << endl;
   else
      cout << "The stack s1 is not greater than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is not greater than the stack s2.
The stack s1 is greater than the stack s3.
```

`operator>=`

Tests if the stack object on the left side of the operator is greater than or equal to the stack object on the right side.

C++

```
bool operator>=(const stack <Type, Container>& left, const stack <Type,
Container>& right);
```

Parameters

left
An object of type `stack`.

right
An object of type `stack`.

Return Value

`true` if the stack on the left side of the operator is strictly less than the stack on the right side of the operator; otherwise `false`.

Remarks

The comparison between stack objects is based on a pairwise comparison of their elements. The greater than or equal to relationship between two stack objects is based on a comparison of the first pair of unequal elements.

Example

C++

```cpp
// stack_op_ge.cpp
// compile with: /EHsc
#include <stack>
#include <list>
#include <iostream>

int main( )
{
   using namespace std;

   // Declares stacks with list base container
   stack <int, list<int> > s1, s2, s3;

   s1.push( 1 );
   s1.push( 2 );
   s2.push( 5 );
   s2.push( 10 );
   s3.push( 1 );
   s3.push( 2 );

   if ( s1 >= s2 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s2." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s2." << endl;

   if ( s1>= s3 )
      cout << "The stack s1 is greater than or equal to "
           << "the stack s3." << endl;
   else
      cout << "The stack s1 is less than "
           << "the stack s3." << endl;
}
```

Output

```
The stack s1 is less than the stack s2.
The stack s1 is greater than or equal to the stack s3.
```

`stack` Class

Article • 02/08/2022

A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type. The `stack` class is used when it is important to be clear that only `stack` operations are being performed on the container.

Syntax

```
C++
```

```
template <class Type, class Container= deque <Type>>
class stack
```

Parameters

`Type`

The element data type to be stored in the stack.

`Container`

The type of the underlying container used to implement the stack. The default value is the class `deque<Type>`.

Remarks

The elements of class `Type` stipulated in the first template parameter of a stack object are synonymous with value_type and must match the type of element in the underlying container class `Container` stipulated by the second template parameter. The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.

Suitable underlying container classes for stack include deque, list class, and vector class, or any other sequence container that supports the operations of `back`, `push_back`, and `pop_back`. The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.

The `stack` objects are equality comparable if and only if the elements of class `Type` are equality comparable and are less-than comparable if and only if the elements of class `Type` are less-than comparable.

- The `stack` class supports a last-in, first-out (LIFO) data structure. A good analogue to keep in mind would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the `stack` class.

- The queue class supports a first-in, first-out (FIFO) data structure. A good analogue to keep in mind would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason fur using the `queue` class.

- The priority_queue class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogue to keep in mind would be people lining up where they are arranged by age, height, or some other criterion.

Members

Constructors

Name	Description
stack	Constructs a `stack` that is empty or that is a copy of a base container object.

Typedefs

Name	Description
container_type	A type that provides the base container to be adapted by a `stack`.
size_type	An unsigned integer type that can represent the number of elements in a `stack`.
value_type	A type that represents the type of object stored as an element in a `stack`.

Functions

Name	Description
empty	Tests if the `stack` is empty.
pop	Removes the element from the top of the `stack`.
push	Adds an element to the top of the `stack`.
size	Returns the number of elements in the `stack`.
top	Returns a reference to an element at the top of the `stack`.

`container_type`

A type that provides the base container to be adapted.

```
C++
```

```
typedef Container container_type;
```

Remarks

The type is a synonym for the template parameter `Container`. All three C++ Standard Library sequence container classes — the `vector` class, `list` class, and the default class `deque` — meet the requirements to be used as the base container for a `stack` object. User-defined types satisfying these requirements may also be used.

For more information on `Container`, see the Remarks section of the stack Class topic.

Example

See the example for stack::stack for an example of how to declare and use `container_type`.

`empty`

Tests if a stack is empty.

```
C++
```

```
bool empty() const;
```

Return Value

`true` if the stack is empty; `false` if the stack is nonempty.

Example

C++

```
// stack_empty.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   // Declares stacks with default deque base container
   stack <int> s1, s2;

   s1.push( 1 );

   if ( s1.empty( ) )
      cout << "The stack s1 is empty." << endl;
   else
      cout << "The stack s1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The stack s2 is empty." << endl;
   else
      cout << "The stack s2 is not empty." << endl;
}
```

Output

```
The stack s1 is not empty.
The stack s2 is empty.
```

pop

Removes the element from the top of the stack.

C++

```
void pop();
```

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

C++

```
// stack_pop.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1, s2;

   s1.push( 10 );
   s1.push( 20 );
   s1.push( 30 );

   stack <int>::size_type i;
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   i = s1.top( );
   cout << "The element at the top of the stack is "
        << i << "." << endl;

   s1.pop( );

   i = s1.size( );
   cout << "After a pop, the stack length is "
        << i << "." << endl;

   i = s1.top( );
   cout << "After a pop, the element at the top of the stack is "
        << i << "." << endl;
}
```

Output

```
The stack length is 3.
The element at the top of the stack is 30.
After a pop, the stack length is 2.
After a pop, the element at the top of the stack is 20.
```

push

Adds an element to the top of the stack.

C++

```
void push(const Type& val);
```

Parameters

val

The element added to the top of the stack.

Remarks

The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

Example

C++

```
// stack_push.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1;

   s1.push( 10 );
   s1.push( 20 );
   s1.push( 30 );

   stack <int>::size_type i;
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   i = s1.top( );
   cout << "The element at the top of the stack is "
        << i << "." << endl;
}
```

Output

```
The stack length is 3.
The element at the top of the stack is 30.
```

`size`

Returns the number of elements in the stack.

C++

```
size_type size() const;
```

Return Value

The current length of the stack.

Example

C++

```
// stack_size.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1, s2;
   stack <int>::size_type i;

   s1.push( 1 );
   i = s1.size( );
   cout << "The stack length is " << i << "." << endl;

   s1.push( 2 );
   i = s1.size( );
   cout << "The stack length is now " << i << "." << endl;
}
```

Output

```
The stack length is 1.
The stack length is now 2.
```

`size_type`

An unsigned integer type that can represent the number of elements in a stack.

```
C++
```

```
typedef typename Container::size_type size_type;
```

Remarks

The type is a synonym for `size_type` of the base container adapted by the stack.

Example

See the example for size for an example of how to declare and use `size_type`.

`stack`

Constructs a stack that is empty or that is a copy of a base container class.

```
C++
```

```
stack();

explicit stack(const container_type& right);
```

Parameters

`right`
The container of which the constructed stack is to be a copy.

Example

```
C++
```

```
// stack_stack.cpp
// compile with: /EHsc
#include <stack>
#include <vector>
#include <list>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   // Declares stack with default deque base container
   stack <char> dsc1;

   //Explicitly declares a stack with deque base container
   stack <char, deque<char> > dsc2;

   // Declares a stack with vector base containers
   stack <int, vector<int> > vsi1;

   // Declares a stack with list base container
   stack <int, list<int> > lsi;

   // The second member function copies elements from a container
   vector<int> v1;
   v1.push_back( 1 );
   stack <int, vector<int> > vsi2( v1 );
   cout << "The element at the top of stack vsi2 is "
        << vsi2.top( ) << "." << endl;
}
```

Output

```
The element at the top of stack vsi2 is 1.
```

top

Returns a reference to an element at the top of the stack.

C++

```
reference top();

const_reference top() const;
```

Return Value

A reference to the last element in the container at the top of the stack.

Remarks

The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the

end of the container.

If the return value of `top` is assigned to a `const_reference`, the `stack` object cannot be modified. If the return value of `top` is assigned to a `reference`, the `stack` object can be modified.

Example

C++

```
// stack_top.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   stack <int> s1;

   s1.push( 1 );
   s1.push( 2 );

   int& i = s1.top( );
   const int& ii = s1.top( );

   cout << "The top integer of the stack s1 is "
        << i << "." << endl;
   i--;
   cout << "The next integer down is "<< ii << "." << endl;
}
```

Output

```
The top integer of the stack s1 is 2.
The next integer down is 1.
```

`value_type`

A type that represents the type of object stored as an element in a stack.

C++

```
typedef typename Container::value_type value_type;
```

Remarks

The type is a synonym for `value_type` of the base container adapted by the stack.

Example

C++

```
// stack_value_type.cpp
// compile with: /EHsc
#include <stack>
#include <iostream>

int main( )
{
   using namespace std;
   // Declares stacks with default deque base container
   stack<int>::value_type AnInt;

   AnInt = 69;
   cout << "The value_type is AnInt = " << AnInt << endl;

   stack<int> s1;
   s1.push( AnInt );
   cout << "The element at the top of the stack is "
        << s1.top( ) << "." << endl;
}
```

Output

```
The value_type is AnInt = 69
The element at the top of the stack is 69.
```

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

<stdexcept>

Article • 12/07/2021

Defines several standard classes used for reporting exceptions. The classes form a derivation hierarchy all derived from class exception and include two general types of exceptions: logical errors and run-time errors. The logical errors are caused programmer mistakes. They derive from the base class logic_error and include:

- `domain_error`
- `invalid_argument`
- `length_error`
- `out_of_range`

The run-time errors occur because of mistakes in either the library functions or in the run-time system. They derive from the base class runtime_error and include:

- `overflow_error`
- `range_error`
- `underflow_error`

Classes

Class	Description
domain_error Class	The class serves as the base class for all exceptions thrown to report a domain error.
invalid_argument Class	The class serves as the base class for all exceptions thrown to report an invalid argument.
length_error Class	The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.
logic_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.
out_of_range Class	The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.

Class	Description
overflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic overflow.
range_error Class	The class serves as the base class for all exceptions thrown to report a range error.
runtime_error Class	The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.
underflow_error Class	The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

domain_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report a domain error (as in mathematics, not networking).

Syntax

C++

```
class domain_error : public logic_error {
public:
    explicit domain_error(const string& message);

    explicit domain_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

`domain_error` isn't thrown by any functions in the Microsoft implementation of the C++ Standard Library, but it might be thrown by third-party libraries or user code.

Example

C++

```
// domain_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      throw domain_error("Your domain is in error!");
   }
```

```
    catch (const exception& e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: Your domain is in error!
Type: class std::domain_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

logic_error Class

Thread Safety in the C++ Standard Library

invalid_argument Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report an invalid argument.

Syntax

C++

```
class invalid_argument : public logic_error {
public:
    explicit invalid_argument(const string& message);

    explicit invalid_argument(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// invalid_argument.cpp
// compile with: /EHsc
#include <bitset>
#include <exception>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      bitset<32> b("11001010101100001b1001010101110000");
   }
   catch (const exception& e)
   {
      cerr << "Caught: " << e.what() << endl;
```

```
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: invalid bitset char
Type: class std::invalid_argument
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

logic_error Class

Thread Safety in the C++ Standard Library

length_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.

Syntax

C++

```
class length_error : public logic_error {
public:
    explicit length_error(const string& message);

    explicit length_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// length_error.cpp
// compile with: /EHsc
#include <cstddef>
#include <exception>
#include <iostream>
#include <typeinfo>
#include <vector>
using namespace std;

int main()
{
   try
   {
      vector<int> v(100 + static_cast<size_t>(-1) / sizeof(int));
   }
   catch (const exception& e)
   {
```

```
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: vector too long
Type: class std::length_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

logic_error Class

Thread Safety in the C++ Standard Library

logic_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.

Syntax

C++

```
class logic_error : public exception {
public:
    explicit logic_error(const string& message);

    explicit logic_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// logic_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      throw logic_error("Does not compute!");
   }
   catch (const exception& e)
   {
      cerr << "Caught: " << e.what() << endl;
```

```
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: Does not compute!
Type: class std::logic_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

exception Class

Thread Safety in the C++ Standard Library

out_of_range Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range.

Syntax

C++

```
class out_of_range : public logic_error {
public:
    explicit out_of_range(const string& message);

    explicit out_of_range(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// out_of_range.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      string str("Micro");
      string rstr("soft");
      str.append(rstr, 5, 3);
      cout << str << endl;
   }
```

```
    catch (const exception& e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: invalid string position
Type: class std::out_of_range
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

logic_error Class

Thread Safety in the C++ Standard Library

overflow_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report an arithmetic overflow.

Syntax

C++

```
class overflow_error : public runtime_error {
public:
    explicit overflow_error(const string& message);

    explicit overflow_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// overflow_error.cpp
// compile with: /EHsc
#include <bitset>
#include <exception>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      bitset<33> b;
      b[32] = 1;
      b[0] = 1;
      unsigned long x = b.to_ulong();
   }
```

```
    catch (const exception& e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: bitset overflow
Type: class std::overflow_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

runtime_error Class

Thread Safety in the C++ Standard Library

range_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report a range error (as in mathematics, not iterators).

Syntax

C++

```
class range_error : public runtime_error {
public:
    explicit range_error(const string& message);
    explicit range_error(const char *message);
};
```

Remarks

The value returned by what is a copy of `message.data()`. For more information, see basic_string::data.

Example

C++

```
// range_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      throw range_error("The range is in error!");
   }
   catch (const exception& e)
   {
      cerr << "Caught: " << e.what() << endl;
      cerr << "Type: " << typeid(e).name() << endl;
   }
```

```
}
/* Output:
Caught: The range is in error!
Type: class std::range_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

runtime_error Class

Thread Safety in the C++ Standard Library

runtime_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes.

Syntax

C++

```
class runtime_error : public exception {
public:
    explicit runtime_error(const string& message);

    explicit runtime_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

Example

C++

```
// runtime_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <locale>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      locale loc("test");
   }
   catch (const exception& e)
   {
      cerr << "Caught: " << e.what() << endl;
```

```
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: bad locale name
Type: class std::runtime_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

exception Class

Thread Safety in the C++ Standard Library

underflow_error Class

Article • 09/13/2021

The class serves as the base class for all exceptions thrown to report an arithmetic underflow.

Syntax

C++

```
class underflow_error : public runtime_error {
public:
    explicit underflow_error(const string& message);

    explicit underflow_error(const char *message);

};
```

Remarks

The value returned by `what()` is a copy of `message.data()`. For more information, see what and data.

`underflow_error` isn't thrown by any functions in the Microsoft implementation of the C++ Standard Library, but it might be thrown by third-party libraries or user code.

Example

C++

```
// underflow_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      throw underflow_error("The number's a bit small, captain!");
   }
```

```
    catch (const exception& e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: The number's a bit small, captain!
Type: class std::underflow_error
*/
```

Requirements

Header: <stdexcept>

Namespace: std

See also

runtime_error Class

Thread Safety in the C++ Standard Library

<streambuf>

Article • 12/07/2021

Include the iostreams standard header <streambuf> to define the class template basic_streambuf, which is basic to the operation of the iostreams classes. This header is typically included for you by another of the iostreams headers; you rarely need to include it directly.

Syntax

```
C++

#include <streambuf>
```

Typedefs

Type name	Description
streambuf	A specialization of `basic_streambuf` that uses `char` as the template parameters.
wstreambuf	A specialization of `basic_streambuf` that uses `wchar_t` as the template parameters.

Classes

Class	Description
basic_streambuf Class	The class template describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

`<streambuf>` typedefs

Article • 12/07/2021

streambuf
wstreambuf

streambuf

A specialization of `basic_streambuf` that uses `char` as the template parameters.

```
C++
```

```
typedef basic_streambuf<char, char_traits<char>> streambuf;
```

Remarks

The type is a synonym for the class template basic_streambuf, specialized for elements of type `char` with default character traits.

wstreambuf

A specialization of `basic_streambuf` that uses `wchar_t` as the template parameters.

```
C++
```

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t>> wstreambuf;
```

Remarks

The type is a synonym for the class template basic_streambuf, specialized for elements of type `wchar_t` with default character traits.

See also

<streambuf>

basic_streambuf Class

Article • 06/16/2022

Describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream.

Syntax

```
C++
```

```
template <class Elem, class Tr = char_traits<Elem>>
class basic_streambuf;
```

Parameters

Elem
A char_type.

Tr
The character traits_type.

Remarks

The class template describes an abstract base class for deriving a stream buffer, which controls the transmission of elements to and from a specific representation of a stream. An object of class `basic_streambuf` helps control a stream with elements of type *Tr*, also known as char_type, whose character traits are determined by the class char_traits, also known as traits_type.

Every stream buffer conceptually controls two independent streams: one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a basic_stringbuf< `Elem`, `Tr` > object, for example, is what you later extract from its input stream. When you position one stream of a basic_filebuf< `Elem`, `Tr` > object, you position the other stream in tandem.

The public interface to class template `basic_streambuf` supplies the operations that are common to all stream buffers, however specialized. The protected interface supplies the

operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each derived stream buffer in this library describes how it specializes the behavior of its protected virtual member functions. The default behavior for the base class, which is often to do nothing, is described in this article.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An input buffer, for example, is characterized by:

- eback, a pointer to the beginning of the buffer.
- gptr, a pointer to the next element to read.
- egptr, a pointer past the end of the buffer.

Similarly, an output buffer is characterized by:

- pbase, a pointer to the beginning of the buffer.
- pptr, a pointer to the next element to write.
- epptr, a pointer past the end of the buffer.

For any buffer, the following protocol is used:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. They can be safely compared for order.
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the write position designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the read position designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the putback position designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from `basic_streambuf<Elem, Tr>` must cooperate in maintaining this protocol.

An object of class `basic_streambuf<Elem, Tr>` stores the six pointers previously described. It also stores a locale object in an object of type locale for potential use by a derived stream buffer.

Constructors

Constructor	Description
basic_streambuf	Constructs an object of type `basic_streambuf`.

Typedefs

Type name	Description
char_type	Associates a type name with the `Elem` template parameter.
int_type	Associates a type name within `basic_streambuf` scope with the `Elem` template parameter.
off_type	Associates a type name within `basic_streambuf` scope with the `Elem` template parameter.
pos_type	Associates a type name within `basic_streambuf` scope with the `Elem` template parameter.
traits_type	Associates a type name with the `Tr` template parameter.

Member functions

Member function	Description
eback	A protected function that returns a pointer to the beginning of the input buffer.
egptr	A protected function that returns a pointer past the end of the input buffer.
epptr	A protected function that returns a pointer past the end of the output buffer.
gbump	A protected function that adds `count` to the next pointer for the input buffer.
getloc	Gets the `basic_streambuf` object's locale.
gptr	A protected function that returns a pointer to the next element of the input buffer.
imbue	A protected, virtual function called by pubimbue.
in_avail	Returns the number of elements that are ready to be read from the buffer.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.

Member function	Description
pbackfail	A protected virtual member function that tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).
pbase	A protected function that returns a pointer to the beginning of the output buffer.
pbump	A protected function that adds `count` to the next pointer for the output buffer.
pptr	A protected function that returns a pointer to the next element of the output buffer.
pubimbue	Sets the `basic_streambuf` object's locale.
pubseekoff	Calls seekoff, a protected virtual function that is overridden in a derived class.
pubseekpos	Calls seekpos, a protected virtual function that is overridden in a derived class and resets the current pointer position.
pubsetbuf	Calls setbuf, a protected virtual function that is overridden in a derived class.
pubsync	Calls sync, a protected virtual function that is overridden in a derived class and updates the external stream associated with this buffer.
sbumpc	Reads and returns the current element, moving the stream pointer.
seekoff	The protected virtual member function tries to alter the current positions for the controlled streams.
seekpos	The protected virtual member function tries to alter the current positions for the controlled streams.
setbuf	The protected virtual member function performs an operation particular to each derived stream buffer.
setg	A protected function that stores `_Gbeg` in the beginning pointer, `_Gnext` in the next pointer, and `_Gend` in the end pointer for the input buffer.
setp	A protected function that stores `_Pbeg` in the beginning pointer and `_Pend` in the end pointer for the output buffer.
sgetc	Returns current element without changing position in the stream.
sgetn	Returns the number of elements read.
showmanyc	Protected virtual member function that returns a count of the number of characters that can be extracted from the input stream. It also ensures that the program won't be subject to an indefinite wait.
snextc	Reads the current element and returns the following element.

Member function	Description
sputbackc	Puts a `char_type` in the stream.
sputc	Puts a character into the stream.
sputn	Puts a character string into the stream.
stossc	Move past the current element in the stream.
sungetc	Gets a character from the stream.
swap	Exchanges the values in this object for the values in the provided `basic_streambuf` object parameter.
sync	A protected virtual function that tries to synchronize the controlled streams with any associated external streams.
uflow	A protected virtual function that extracts the current element from the input stream.
underflow	A protected virtual function that extracts the current element from the input stream.
xsgetn	A protected virtual function that extracts elements from the input stream.
xsputn	A protected virtual function that inserts elements into the output stream.

Operators

Operator	Description
operator=	Assigns the values of this object from another `basic_streambuf` object.

Requirements

Header: <streambuf>

Namespace: std

basic_streambuf::basic_streambuf

Constructs an object of type `basic_streambuf`.

```
C++
```

```
basic_streambuf();

basic_streambuf(const basic_streambuf& right);
```

Parameters

right
An lvalue reference to the `basic_streambuf` object that is used to set the values for this `basic_streambuf` object.

Remarks

The first protected constructor stores a null pointer in all pointers controlling the input buffer and the output buffer. It also stores `locale::classic` in the locale object. For more information, see locale::classic.

The second protected constructor copies the pointers and locale from *right*.

basic_streambuf::char_type

Associates a type name with the **Elem** template parameter.

```
C++

typedef Elem char_type;
```

basic_streambuf::eback

A protected function that returns a pointer to the beginning of the input buffer.

```
C++

char_type *eback() const;
```

Return Value

A pointer to the beginning of the input buffer.

basic_streambuf::egptr

A protected function that returns a pointer past the end of the input buffer.

```
C++
```

```
char_type *egptr() const;
```

Return Value

A pointer past the end of the input buffer.

basic_streambuf::epptr

A protected function that returns a pointer past the end of the output buffer.

```
C++
```

```
char_type *epptr() const;
```

Return Value

A pointer past the end of the output buffer.

basic_streambuf::gbump

A protected function that adds *count* to the next pointer for the input buffer.

```
C++
```

```
void gbump(int count);
```

Parameters

count
The amount by which to advance the pointer.

basic_streambuf::getloc

Gets the basic_streambuf object's locale.

```
C++
```

```
locale getloc() const;
```

Return Value

The stored locale object.

Remarks

For related information, see ios_base::getloc.

Example

C++

```
// basic_streambuf_getloc.cpp
// compile with: /EHsc
#include <iostream>

int main( )
{
   using namespace std;
   cout << cout.rdbuf( )->getloc( ).name( ).c_str( ) << endl;
}
```

Output

```
C
```

basic_streambuf::gptr

A protected function that returns a pointer to the next element of the input buffer.

C++

```
char_type *gptr() const;
```

Return Value

A pointer to the next element of the input buffer.

basic_streambuf::imbue

A protected virtual function called by pubimbue.

```
C++
```

```
virtual void imbue(const locale& _Loc);
```

Parameters

_Loc
A reference to a locale.

Remarks

The default behavior is to do nothing.

basic_streambuf::in_avail

Returns the number of elements that are ready to be read from the buffer.

```
C++
```

```
streamsize in_avail();
```

Return Value

The number of elements that are ready to be read from the buffer.

Remarks

If a read position is available, the member function returns egptr - gptr. Otherwise, it returns showmanyc.

Example

```
C++
```

```
// basic_streambuf_in_avail.cpp
// compile with: /EHsc
#include <iostream>
```

```
int main( )
{
   using namespace std;
   char c;
   // cin's buffer is empty, in_avail will return 0
   cout << cin.rdbuf( )->in_avail( ) << endl;
   cin >> c;
   cout << cin.rdbuf( )->in_avail( ) << endl;
}
```

basic_streambuf::int_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

C++

```
typedef typename traits_type::int_type int_type;
```

basic_streambuf::off_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

C++

```
typedef typename traits_type::off_type off_type;
```

basic_streambuf::operator=

Assigns the values of this object from another `basic_streambuf` object.

C++

```
basic_streambuf& operator=(const basic_streambuf& right);
```

Parameters

right
An lvalue reference to the `basic_streambuf` object that is used to assign values to this

object.

Remarks

The protected member operator copies from *right* the pointers that control the input buffer and the output buffer. It also stores `right.`getloc() in the `locale object`. It returns `*this`.

basic_streambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

```
C++
```

```
virtual int_type overflow(int_type _Meta = traits_type::eof());
```

Parameters

_Meta
The character to insert into the buffer, or **traits_type::**eof.

Return Value

If the function can't succeed, it returns **traits_type::eof** or throws an exception. Otherwise, it returns **traits_type::**not_eof(_ *Meta*). The default behavior is to return **traits_type::eof**.

Remarks

If *_Meta* doesn't compare equal to **traits_type::eof**, the protected virtual member function endeavors to insert the element **traits_type::**to_char_type(*_Meta*) into the output stream. It can do so in various ways:

- If a `write position` is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.

- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

The virtual overflow function, together with the sync and underflow functions, defines the characteristics of the streambuf-derived class. Each derived class might implement overflow differently, but the interface with the calling stream class is the same.

The `overflow` function is most frequently called by public `streambuf` functions like `sputc` and `sputn` when the put area is full, but other classes, including the stream classes, can call `overflow` anytime.

The function consumes the characters in the put area between the `pbase` and `pptr` pointers and then reinitializes the put area. The `overflow` function must also consume `nCh` (if `nCh` isn't `EOF`), or it might choose to put that character in the new put area so that it will be consumed on the next call.

The definition of consume varies among derived classes. For example, the `filebuf` class writes its characters to a file, while the `strstreambuf` class keeps them in its buffer and (if the buffer is designated as dynamic) expands the buffer in response to a call to overflow. This expansion is achieved by freeing the old buffer and replacing it with a new, larger one. The pointers are adjusted as necessary.

basic_streambuf::pbackfail

A protected virtual member function that tries to put back an element into the input stream, then make it the current element (pointed to by the next pointer).

```
C++

virtual int_type pbackfail(int_type _Meta = traits_type::eof());
```

Parameters

_Meta
The character to insert into the buffer, or **traits_type::**eof.

Return Value

If the function can't succeed, it returns **traits_type::eof** or throws an exception. Otherwise, it returns some other value. The default behavior is to return **traits_type::eof**.

Remarks

If *_Meta* compares equal to **traits_type::eof**, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by **traits_type::**to_char_type(*_Meta*). The function can put back an element in various ways:

- If a putback position is available, it can store the element into the putback position and decrement the next pointer for the input buffer.
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

basic_streambuf::pbase

A protected function that returns a pointer to the beginning of the output buffer.

```
C++

char_type *pbase() const;
```

Return Value

A pointer to the beginning of the output buffer.

basic_streambuf::pbump

A protected function that adds *count* to the next pointer for the output buffer.

```
C++

void pbump(int count);
```

Parameters

count
The number of characters by which to move the write position forward.

basic_streambuf::pos_type

Associates a type name within basic_streambuf scope with one of the types in a template parameter.

```
C++
```

```
typedef typename traits_type::pos_type pos_type;
```

basic_streambuf::pptr

A protected function that returns a pointer to the next element of the output buffer.

```
C++
```

```
char_type *pptr() const;
```

Return Value

A pointer to the next element of the output buffer.

basic_streambuf::pubimbue

Sets the basic_streambuf object's locale.

```
C++
```

```
locale pubimbue(const locale& _Loc);
```

Parameters

_Loc
A reference to a locale.

Return Value

The previous value stored in the locale object.

Remarks

The member function stores _ *Loc* in the locale object and calls imbue.

Example

See basic_ios::imbue for an example that uses `pubimbue`.

basic_streambuf::pubseekoff

Calls seekoff, a protected virtual function that is overridden in a derived class.

```
C++
```

```
pos_type pubseekoff(off_type _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off
The position to seek for relative to *_Way*.

_Way
The starting point for offset operations. See seekdir for possible values.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position (seekoff(_ *Off*, `_Way`, `_Which`)).

Remarks

Moves the pointer relative to *_Way*.

basic_streambuf::pubseekpos

Calls seekpos, a protected virtual function that is overridden in a derived class, and resets the current pointer position.

```
C++
```

```
pos_type pubseekpos(pos_type _Sp, ios_base::openmode _Which = ios_base::in |
ios_base::out);
```

Parameters

_Sp
The position to seek for.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The member function returns seekpos(_ *Sp*, `_Which`).

basic_streambuf::pubsetbuf

Calls setbuf, a protected virtual function that is overridden in a derived class.

```
C++
```

```
basic_streambuf<Elem, Tr> *pubsetbuf(
    char_type* _Buffer,
    streamsize count);
```

Parameters

_Buffer
A pointer to `char_type` for this instantiation.

count
The size of the buffer.

Return Value

Returns setbuf(`_Buffer`, `count`).

basic_streambuf::pubsync

Calls sync, a protected virtual function that is overridden in a derived class, and updates the external stream associated with this buffer.

```
C++
```

```
int pubsync();
```

Return Value

Returns sync or -1 if failure.

basic_streambuf::sbumpc

Reads and returns the current element, moving the stream pointer.

```
C++
```

```
int_type sbumpc();
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::**to_int_type(*gptr) and increments the next pointer for the input buffer. Otherwise, it returns uflow.

Example

```
C++
```

```
// basic_streambuf_sbumpc.cpp
// compile with: /EHsc
```

```
#include <iostream>

int main( )
{
   using namespace std;
   int i = 0;
   i = cin.rdbuf( )->sbumpc( );
   cout << i << endl;
}
```

Input

```
3
```

Output

```
33
51
```

basic_streambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

C++

```
virtual pos_type seekoff(
    off_type _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off
The position to seek for relative to *_Way*.

_Way
The starting point for offset operations. See seekdir for possible values.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

Returns the new position or an invalid stream position (`seekoff` (_ *Off*, `_Way`, `_Which`)).

Remarks

The new position is determined as follows:

- If `_Way` == `ios_base::beg`, the new position is the beginning of the stream plus _ *Off*.
- If `_Way` == `ios_base::cur`, the new position is the current stream position plus _ *Off*.
- If `_Way` == `ios_base::end`, the new position is the end of the stream plus _ *Off*.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
C++
```

```
virtual pos_type seekpos(pos_type _Sp, ios_base::openmode _Which =
ios_base::in | ios_base::out);
```

Parameters

_Sp
The position to seek for.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

The new position, or an invalid stream position. To determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The new position is _ *Sp*.

Typically, if **which & ios_base::in** is nonzero, the input stream is affected, and if **which & ios_base::out** is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position or positions, it returns the resulting stream position or one of the resulting stream positions. Otherwise, it returns an invalid stream position (-1). The default behavior is to return an invalid stream position.

basic_streambuf::setbuf

A protected virtual member function that performs an operation particular to each derived stream buffer.

```
C++
```

```
virtual basic_streambuf<Elem, Tr> *setbuf(
    char_type* _Buffer,
    streamsize count);
```

Parameters

_Buffer
Pointer to a buffer.

count
Size of the buffer.

Return Value

The default behavior is to return **`this`**.

Remarks

See basic_filebuf. `setbuf` provides an area of memory for the `streambuf` object to use. How the buffer is used in defined in the derived classes.

basic_streambuf::setg

A protected function that stores _ *Gbeg* in the beginning pointer, `_Gnext` in the next pointer, and `_Gend` in the end pointer for the input buffer.

```
C++
```

```
void setg(char_type* _Gbeg,
    char_type* _Gnext,
    char_type* _Gend);
```

Parameters

_Gbeg
A pointer to the beginning of the buffer.

_Gnext
A pointer to somewhere in the middle of the buffer.

_Gend
A pointer to the end of the buffer.

basic_streambuf::setp

A protected function that stores *_Pbeg* in the beginning pointer and *_Pend* in the end pointer for the output buffer.

```
C++
```

```
void setp(char_type* _Pbeg, char_type* _Pend);
```

Parameters

_Pbeg
A pointer to the beginning of the buffer.

_Pend
A pointer to the end of the buffer.

basic_streambuf::sgetc

Returns current element without changing position in the stream.

```
C++
```

```
int_type sgetc();
```

Return Value

The current element.

Remarks

If a read position is available, the member function returns **traits_type::**to_int_type(`*`gptr). Otherwise, it returns underflow.

Example

```
C++
```

```
// basic_streambuf_sgetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ifstream myfile( "basic_streambuf_sgetc.txt", ios::in );

   char i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
   i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
}
```

basic_streambuf::sgetn

Extracts up to *count* characters from the input buffer and stores them in the provided buffer *ptr*.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
C++
```

```
streamsize sgetn(
    char_type* ptr,
    streamsize count);
```

Parameters

ptr
The buffer to contain the extracted characters.

count
The number of elements to read.

Return Value

The number of elements read. For more information, see streamsize for more information.

Remarks

The member function returns xsgetn(`ptr`, `count`).

Example

```
C++
```

```
// basic_streambuf_sgetn.cpp
// compile with: /EHsc /W3
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    ifstream myfile("basic_streambuf_sgetn.txt", ios::in);
    char a[10];
```

```
    // Extract 3 characters from myfile and store them in a.
    streamsize i = myfile.rdbuf()->sgetn(&a[0], 3);  // C4996
    a[i] = myfile.widen('\0');

    // Display the size and contents of the buffer passed to sgetn.
    cout << i << " " << a << endl;

    // Display the contents of the original input buffer.
    cout << myfile.rdbuf() << endl;
}
```

basic_streambuf::showmanyc

A protected virtual member function that returns a count of the number of characters that can be extracted from the input stream and ensure that the program won't be subject to an indefinite wait.

```
C++

virtual streamsize showmanyc();
```

Return Value

The default behavior is to return zero.

basic_streambuf::snextc

Reads the current element and returns the following element.

```
C++

int_type snextc();
```

Return Value

The next element in the stream.

Remarks

The member function calls sbumpc and, if that function returns **traits_type::**eof, returns **traits_type::eof**. Otherwise, it returns sgetc.

Example

C++

```
// basic_streambuf_snextc.cpp
// compile with: /EHsc
#include <iostream>
int main( )
{
   using namespace std;
   int i = 0;
   i = cin.rdbuf( )->snextc( );
   // cout << ( int )char_traits<char>::eof << endl;
   cout << i << endl;
}
```

Input

```
aa
```

Output

```
aa97
```

basic_streambuf::sputbackc

Puts a char_type in the stream.

C++

```
int_type sputbackc(char_type _Ch);
```

Parameters

_Ch
The character.

Return Value

Returns the character or failure.

Remarks

If a putback position is available and *_Ch* compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns **traits_type::**to_int_type(`_Ch`). Otherwise, it returns pbackfail(`_Ch`).

Example

C++

```
// basic_streambuf_sputbackc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
    using namespace std;

    ifstream myfile("basic_streambuf_sputbackc.txt",
        ios::in);

    int i = myfile.rdbuf()->sbumpc();
    cout << (char)i << endl;
    int j = myfile.rdbuf()->sputbackc('z');
    if (j == 'z')
    {
        cout << "it worked" << endl;
    }
    i = myfile.rdbuf()->sgetc();
    cout << (char)i << endl;
}
```

basic_streambuf::sputc

Puts a character into the stream.

C++

```
int_type sputc(char_type _Ch);
```

Parameters

_Ch
The character.

Return Value

Returns the character, if successful.

Remarks

If a `write position` is available, the member function stores *_Ch* in the write position, increments the next pointer for the output buffer, and returns **traits_type::**to_int_type(`_Ch`). Otherwise, it returns overflow(`_Ch`).

Example

C++

```
// basic_streambuf_sputc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   int i = cout.rdbuf( )->sputc( 'a' );
   cout << endl << ( char )i << endl;
}
```

Output

```
a
a
```

basic_streambuf::sputn

Puts a character string into the stream.

C++

```
streamsize sputn(const char_type* ptr, streamsize count);
```

Parameters

ptr
The character string.

count
The count of characters.

Return Value

The number of characters inserted into the stream.

Remarks

The member function returns xsputn(`ptr`, `count`). For more information, see the Remarks section of this member for more information.

Example

C++

```
// basic_streambuf_sputn.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    streamsize i = cout.rdbuf()->sputn("test", 4);
    cout << endl << i << endl;
}
```

Output

```
test
4
```

basic_streambuf::stossc

Move past the current element in the stream.

C++

```
void stossc();
```

Remarks

The member function calls sbumpc. An implementation isn't required to supply this member function.

Example

```
C++
```

```
// basic_streambuf_stossc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;
   ifstream myfile( "basic_streambuf_stossc.txt", ios::in );

   myfile.rdbuf( )->stossc( );
   char i = myfile.rdbuf( )->sgetc( );
   cout << i << endl;
}
```

basic_streambuf::sungetc

Gets a character from the stream.

```
C++
```

```
int_type sungetc();
```

Return Value

Returns either the character or failure.

Remarks

If a putback position is available, the member function decrements the next pointer for the input buffer and returns `traits_type::`to_int_type(`*`gptr). However, it isn't always possible to determine the last character read so that it can be captured in the state of the current buffer. If this is true, then the function returns pbackfail. To avoid this situation, keep track of the character to put back and call `sputbackc(ch)`, which won't

fail provided you don't call it at the beginning of the stream and you don't try to put back more than one character.

Example

C++

```
// basic_streambuf_sungetc.cpp
// compile with: /EHsc
#include <iostream>
#include <fstream>

int main( )
{
   using namespace std;

   ifstream myfile( "basic_streambuf_sungetc.txt", ios::in );

   // Read and increment
   int i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;

   // Read and increment
   i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;

   // Decrement, read, and do not increment
   i = myfile.rdbuf( )->sungetc( );
   cout << ( char )i << endl;

   i = myfile.rdbuf( )->sungetc( );
   cout << ( char )i << endl;

   i = myfile.rdbuf( )->sbumpc( );
   cout << ( char )i << endl;
}
```

basic_streambuf::swap

Exchanges the values in this object for the values in the provided `basic_streambuf` object.

C++

```
void swap(basic_streambuf& right);
```

Parameters

right
An lvalue reference to the `basic_streambuf` object that is used to exchange values.

Remarks

The protected member function exchanges with *right* all the pointers controlling the `input buffer` and the `output buffer`. It also exchanges `right.`getloc() with the `locale` object.

basic_streambuf::sync

A protected virtual function that tries to synchronize the controlled streams with any associated external streams.

```
C++

virtual int sync();
```

Return Value

If the function can't succeed, it returns -1. The default behavior is to return zero.

Remarks

`sync` involves writing out any elements between the beginning and next pointers for the output buffer. It doesn't involve putting back any elements between the next and end pointers for the input buffer.

basic_streambuf::traits_type

Associates a type name with the **Tr** template parameter.

```
C++

typedef Tr traits_type;
```

basic_streambuf::uflow

A protected virtual function that extracts the current element from the input stream.

```
C++
```

```
virtual int_type uflow();
```

Return Value

The current element.

Remarks

The protected virtual member function tries to extract the current element **ch** from the input stream, then advance the current stream position, and return the element as **traits_type::**to_int_type(**ch**). It can do so in various ways:

- If a read position is available, it takes **ch** as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value **ch**.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function can't succeed, it returns **traits_type::**eof, or throws an exception. Otherwise, it returns the current element `ch` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call underflow and, if that function returns **traits_type::eof**, to return **traits_type::eof**. Otherwise, the function returns the current element **ch** in the input stream, converted as previously described, and advances the next pointer for the input buffer.

basic_streambuf::underflow

Protected, virtual function to extract the current element from the input stream.

```
C++
```

```
virtual int_type underflow();
```

Return Value

The current element.

Remarks

The protected virtual member function endeavors to extract the current element **ch** from the input stream, without advancing the current stream position, and return it as `traits_type::` to_int_type(**ch**). It can do so in various ways:

- If a read position is available, **ch** is the element stored in the read position. For more information, see the Remarks section of the basic_streambuf Class.
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements. For more information, see the Remarks section of the basic_streambuf Class.

If the function can't succeed, it returns `traits_type::` eof `()` or throws an exception. Otherwise, it returns the current element in the input stream, converted as previously described. The default behavior is to return `traits_type::eof()`.

The virtual `underflow` function, with the sync and overflow functions, defines the characteristics of the `streambuf`-derived class. Each derived class might implement `underflow` differently, but the interface with the calling stream class is the same.

The `underflow` function is most frequently called by public `streambuf` functions like sgetc and sgetn when the get area is empty, but other classes, including the stream classes, can call `underflow` anytime.

The `underflow` function supplies the get area with characters from the input source. If the get area contains characters, `underflow` returns the first character. If the get area is empty, it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then `underflow` returns `EOF` and leaves the get area empty.

In the `strstreambuf` class, `underflow` adjusts the egptr pointer to access storage that was dynamically allocated by a call to `overflow`.

basic_streambuf::xsgetn

Protected, virtual function to extract elements from the input stream.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct.

```
C++
```

```
virtual streamsize xsgetn(
    char_type* ptr,
    streamsize count);
```

Parameters

ptr
The buffer to contain the extracted characters.

count
The number of elements to extract.

Return Value

The number of elements extracted.

Remarks

The protected virtual member function extracts up to *count* elements from the input stream, as if by repeated calls to sbumpc, and stores them in the array beginning at *ptr*. It returns the number of elements extracted.

basic_streambuf::xsputn

Protected, virtual function to insert elements into the output stream.

```
C++
```

```
virtual streamsize xsputn(const char_type* ptr, streamsize count);
```

Parameters

ptr
Pointer to elements to insert.

count
Number of elements to insert.

Return Value

The number of elements inserted into the stream.

Remarks

The protected virtual member function inserts up to *count* elements into the output stream, as if by repeated calls to sputc, from the array beginning at *ptr*. The insertion of characters into the output stream stops once all *count* characters have been written, or if calling `sputc( count)` would return `traits::eof()`. It returns the number of elements inserted.

See also

Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

<string>

Article • 12/06/2021

Defines the container class template `basic_string` and various supporting templates.

For more information about `basic_string`, see basic_string Class

Syntax

```
C++

#include <string>
```

Remarks

The C++ language and the C++ Standard Library support two types of strings:

- Null-terminated character arrays often referred to as C strings.
- class template objects, of type `basic_string`, that handle all `char`-like template arguments.

Typedefs

Type name	Description
string	A type that describes a specialization of the class template `basic_string` with elements of type `char` as a `string`.
wstring	A type that describes a specialization of the class template `basic_string` with elements of type `wchar_t` as a `wstring`.
u16string	A type that describes a specialization of the class template `basic_string` based on elements of type `char16_t`.
u32string	A type that describes a specialization of the class template `basic_string` based on elements of type `char32_t`.

Operators

Operator	Description
operator+	Concatenates two string objects.
operator!=	Tests if the string object on the left side of the operator is not equal to the string object on the right side.
operator==	Tests if the string object on the left side of the operator is equal to the string object on the right side.
operator<	Tests if the string object on the left side of the operator is less than to the string object on the right side.
operator<=	Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.
operator<<	A template function that inserts a string into the output stream.
operator>	Tests if the string object on the left side of the operator is greater than to the string object on the right side.
operator>=	Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.
operator>>	A template function that extracts a string from the input stream.

Specialized Template Functions

Name	Description
`hash`	Produces a hash of a string.
swap	Exchanges the arrays of characters of two strings.
stod	Converts a character sequence to a `double`.
stof	Converts a character sequence to a `float`.
stoi	Converts a character sequence to an `int`.
stold	Converts a character sequence to a `long double`.
stoll	Converts a character sequence to a `long long`.
stoul	Converts a character sequence to an `unsigned long`.
stoull	Converts a character sequence to an `unsigned long long`.
to_string	Converts a value to a `string`.

Name	Description
to_wstring	Converts a value to a wide string.

Functions

Function	Description
getline Template	Extract a `string`, line by line, from the input stream.

Classes

Class	Description
basic_string Class	A class template that describes objects that can store a sequence of arbitrary character-like objects.
char_traits Struct	A class template that describes attributes associated with a character of type `CharType`

Specializations

Name	Description
char_traits<char> Struct	A struct that is a specialization of the template struct `char_traits<CharType>` to an element of type **`char`**.
char_traits<wchar_t> Struct	A struct that is a specialization of the template struct `char_traits<CharType>` to an element of type **`wchar_t`**.
char_traits<char16_t> Struct	A struct that is a specialization of the template struct `char_traits<CharType>` to an element of type **`char16_t`**.
char_traits<char32_t> Struct	A struct that is a specialization of the template struct `char_traits<CharType>` to an element of type **`char32_t`**.

Requirements

- **Header:** `<string>`
- **Namespace:** std

See also

Header Files Reference

Thread Safety in the C++ Standard Library

`<string>` functions

Article • 12/06/2021

getline
stod
stof
stoi
stol
stold
stoll
stoul
stoull
swap
to_string
to_wstring

`getline`

Extract strings from the input stream line-by-line.

C++

```
// (1) delimiter as parameter
template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>& in_stream,
    basic_string<CharType, Traits, Allocator>& str,
    CharType delimiter);

template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>&& in_stream,
    basic_string<CharType, Traits, Allocator>& str,
    const CharType delimiter);

// (2) default delimiter used
template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& getline(
    basic_istream<CharType, Traits>& in_stream,
    basic_string<CharType, Traits, Allocator>& str);

template <class Allocator, class Traits, class Allocator>
basic_istream<Allocator, Traits>& getline(
    basic_istream<Allocator, Traits>&& in_stream,
    basic_string<Allocator, Traits, Allocator>& str);
```

Parameters

`in_stream`
The input stream from which a string is to be extracted.

`str`
The string into which are read the characters from the input stream.

`delimiter`
The line delimiter.

Return Value

The input stream `in_stream`.

Remarks

The pair of function signatures marked `(1)` extract characters from `in_stream` until `delimiter` is found, storing them in `str`.

The pair of function signatures marked `(2)` use newline as the default line delimiter and behave as `getline(in_stream, str, in_stream. widen('\n'))`.

The second function of each pair is an analog to the first one to support rvalue references.

Extraction stops when one of the following occurs:

- At end of file, in which case the internal state flag of `in_stream` is set to `ios_base::eofbit`.
- After the function extracts an element that compares equal to `delimiter`. The element doesn't get put back or appended to the controlled sequence.
- After the function extracts str.max_size elements. The internal state flag of `in_stream` is set to `ios_base::failbit`.
- Some other error other than the ones previously listed; the internal state flag of `in_stream` is set to `ios_base::badbit`.

For information about internal state flags, see ios_base::iostate.

If the function extracts no elements, the internal state flag of *in_stream* is set to ios_base::failbit. In any case, getline returns *in_stream*.

If an exception is thrown, *in_stream* and *str* are left in a valid state.

Example

The following code demonstrates getline() in two modes: first with the default delimiter (newline) and second with a whitespace as delimiter. The end-of-file character (CTRL-Z on the keyboard) is used to control termination of the while loops. This value sets the internal state flag of cin to eofbit, which must be cleared with basic_ios::clear() before the second while loop will work properly.

C++

```
// compile with: /EHsc /W4
#include <string>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    string str;
    vector<string> v1;
    cout << "Enter a sentence, press ENTER between sentences. (Ctrl-Z to
stop): " << endl;
    // Loop until end-of-file (Ctrl-Z) is input, store each sentence in a
vector.
    // Default delimiter is the newline character.
    while (getline(cin, str)) {
        v1.push_back(str);
    }

    cout << "The following input was stored with newline delimiter:" <<
endl;
    for (const auto& p : v1) {
        cout << p << endl;
    }

    cin.clear();

    vector<string> v2;
    // Now try it with a whitespace delimiter
    while (getline(cin, str, ' ')) {
        v2.push_back(str);
    }

    cout << "The following input was stored with whitespace as delimiter:"
```

```
<< endl;
    for (const auto& p : v2) {
        cout << p << endl;
    }
}
```

stod

Converts a character sequence to a **double**.

C++

```
double stod(
    const string& str,
    size_t* idx = 0);

double stod(
    const wstring& str,
    size_t* idx = 0
);
```

Parameters

str

The character sequence to be converted.

idx

The index value of the first unconverted character.

Return Value

The **double** value.

Remarks

The function converts the sequence of elements in *str* to a value of type **double** as if by calling `strtod( str.c_str(), _Eptr)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

`stof`

Converts a character sequence to a float.

C++

```
float stof(
    const string& str,
    size_t* idx = 0);

float stof(
    const wstring& str,
    size_t* idx = 0);
```

Parameters

`str`

The character sequence to be converted.

`idx`

The index value of the first unconverted character.

Return Value

The **`float`** value.

Remarks

The function converts the sequence of elements in *`str`* to a value of type **`float`** as if by calling `strtof( str.c_str(), _Eptr)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *`idx`* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

`stoi`

Converts a character sequence to an integer.

C++

```
int stoi(
    const string& str,
```

```
    size_t* idx = 0,
    int base = 10);

int stoi(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Return Value

The integer value.

Parameters

str

The character sequence to be converted.

idx

The index value of the first unconverted character.

base

The number base to use.

Remarks

The function `stoi` converts the sequence of characters in *str* to a value of type **`int`** and returns the value. For example, when passed a character sequence "10", the value returned by `stoi` is the integer 10.

`stoi` behaves similarly to the function `strtol` for single-byte characters when it's called in the manner `strtol( str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function; or `wcstol` for wide characters, when it's called in similar manner, `wcstol(Str.c_str(), _Eptr, idx)`. For more information, see strtol, wcstol, _strtol_l, _wcstol_l.

If `str.c_str() == *_Eptr`, `stoi` throws an object of type `invalid_argument`. If such a call would set `errno`, or if the returned value can't be represented as an object of type **`int`**, it throws an object of type `out_of_range`. Otherwise, if *idx* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx`.

`stol`

Converts a character sequence to a **`long`**.

```
C++
```

```
long stol(
    const string& str,
    size_t* idx = 0,
    int base = 10);

long stol(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

`str`

The character sequence to be converted.

`idx`

The index value of the first unconverted character.

`base`

The number base to use.

Return Value

The long-integer value.

Remarks

The function converts the sequence of elements in *str* to a value of type **`long`** as if by calling `strtol( str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *`idx`* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

`stold`

Converts a character sequence to a **`long double`**.

```
C++
```

```
double stold(
    const string& str,
    size_t* idx = 0);

double stold(
    const wstring& str,
    size_t* idx = 0);
```

Parameters

str

The character sequence to be converted.

idx

The index value of the first unconverted character.

Return Value

The **long double** value.

Remarks

The function converts the sequence of elements in *str* to a value of type **long double** as if by calling strtold(str.c_str(), _Eptr), where _Eptr is an object internal to the function. If str.c_str() == *_Eptr, it throws an object of type invalid_argument. If such a call would set errno, it throws an object of type out_of_range. Otherwise, if *idx* isn't a null pointer, the function stores *_Eptr - str.c_str() in *idx and returns the value.

stoll

Converts a character sequence to a **long long**.

C++

```
long long stoll(
    const string& str,
    size_t* idx = 0,
    int base = 10);

long long stoll(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

`str`

The character sequence to be converted.

`idx`

The index value of the first unconverted character.

`base`

The number base to use.

Return Value

The `long long` value.

Remarks

The function converts the sequence of elements in *str* to a value of type `long long` as if by calling `strtoll( str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

`stoul`

Converts a character sequence to an unsigned long.

C++

```
unsigned long stoul(
    const string& str,
    size_t* idx = 0,
    int base = 10);

unsigned long stoul(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

str

The character sequence to be converted.

idx

The index value of the first unconverted character.

base

The number base to use.

Return Value

The unsigned long-integer value.

Remarks

The function converts the sequence of elements in *str* to a value of type **unsigned long** as if by calling `strtoul( str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

stoull

Converts a character sequence to an **unsigned long long**.

```
C++
```

```
unsigned long long stoull(
    const string& str,
    size_t* idx = 0,
    int base = 10);

unsigned long long stoull(
    const wstring& str,
    size_t* idx = 0,
    int base = 10);
```

Parameters

str

The character sequence to be converted.

idx

The index value of the first unconverted character.

base

The number base to use.

Return Value

The **unsigned long long** value.

Remarks

The function converts the sequence of elements in *str* to a value of type **unsigned long long** as if by calling `strtoull( str.c_str(), _Eptr, idx)`, where `_Eptr` is an object internal to the function. If `str.c_str() == *_Eptr`, it throws an object of type `invalid_argument`. If such a call would set `errno`, it throws an object of type `out_of_range`. Otherwise, if *idx* isn't a null pointer, the function stores `*_Eptr - str.c_str()` in `*idx` and returns the value.

swap

Exchanges the arrays of characters of two strings.

C++

```
template <class Traits, class Allocator>
void swap(basic_string<CharType, Traits, Allocator>& left,
basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left

One string whose elements are to be swapped with the elements of another string.

right

The other string whose elements are to be swapped with the first string.

Remarks

The template function executes the specialized member function left.swap(*right*) for strings, which guarantees constant complexity.

Example

C++

```
// string_swap.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an object of type basic_string<char>
   string s1 ( "Tweedledee" );
   string s2 ( "Tweedledum" );
   cout << "Before swapping string s1 and s2:" << endl;
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   swap ( s1 , s2 );
   cout << "\nAfter swapping string s1 and s2:" << endl;
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;
}
```

Output

```
Before swapping string s1 and s2:
The basic_string s1 = Tweedledee.
The basic_string s2 = Tweedledum.

After swapping string s1 and s2:
The basic_string s1 = Tweedledum.
The basic_string s2 = Tweedledee.
```

`to_string`

Converts a value to a `string`.

C++

```
string to_string(int value);
string to_string(unsigned int value);
string to_string(long value);
```

```
string to_string(unsigned long value);
string to_string(long long value);
string to_string(unsigned long long value);
string to_string(float value);
string to_string(double value);
string to_string(long double value);
```

Parameters

`value`
The value to be converted.

Return Value

The `string` that represents the value.

Remarks

The function converts *value* to a sequence of elements stored in an array object `Buf` internal to the function as if by calling `sprintf(Buf, Fmt, value)`, where `Fmt` is

- `"%d"` if `value` is of type `int`
- `"%u"` if `value` is of type `unsigned int`
- `"%ld"` if `value` is of type `long`
- `"%lu"` if `value` is of type `unsigned long`
- `"%lld"` if `value` is of type `long long`
- `"%llu"` if `value` is of type `unsigned long long`
- `"%f"` if `value` is of type `float` or `double`
- `"%Lf"` if `value` is of type `long double`

The function returns `string(Buf)`.

`to_wstring`

Converts a value to a wide string.

```
```

C++

```
wstring to_wstring(int value);
wstring to_wstring(unsigned int value);
wstring to_wstring(long value);
wstring to_wstring(unsigned long value);
wstring to_wstring(long long value);
wstring to_wstring(unsigned long long value);
wstring to_wstring(float value);
wstring to_wstring(double value);
wstring to_wstring(long double value);
```

Parameters

value

The value to be converted.

Return Value

The wide string that represents the value.

Remarks

The function converts *value* to a sequence of elements stored in an array object `Buf` internal to the function as if by calling `swprintf(Buf, Len, Fmt, value)`, where `Fmt` is

- `L"%d"` if *value* is of type **int**
- `L"%u"` if *value* is of type **unsigned int**
- `L"%ld"` if *value* is of type **long**
- `L"%lu"` if *value* is of type **unsigned long**
- `L"%lld"` if *value* is of type **long long**
- `L"%llu"` if *value* is of type **unsigned long long**
- `L"%f"` if *value* is of type **float** or **double**
- `L"%Lf"` if *value* is of type **long double**

The function returns `wstring(Buf)`.

See also

<string>

`<string>` operators

Article • 12/07/2021

operator!=
operator>
operator>>
operator>=
operator<
operator<<
operator<=
operator+
operator==

operator+

Concatenates two string objects.

C++

```
template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator> operator+(
    const CharType left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>& left,
```

```
    const basic_string<CharType, Traits, Allocator>&& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    const basic_string<CharType, Traits, Allocator>&& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const basic_string<CharType, Traits, Allocator>&& left,
    CharType right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>&& right);

template <class CharType, class Traits, class Allocator>
basic_string<CharType, Traits, Allocator>&& operator+(
    CharType left,
    const basic_string<CharType, Traits, Allocator>&& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be concatenated.

right
A C-style string or an object of type `basic_string` to be concatenated.

Return Value

The string that is the concatenation of the input strings.

Remarks

The functions each overload `operator+` to concatenate two objects of class template basic_string Class. All effectively return `basic_string< CharType, Traits, Allocator>`

`(Left).append(right)`. For more information, see append.

Example

C++

```
// string_op_con.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an object of type basic_string<char>
   string s1 ( "anti" );
   string s2 ( "gravity" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "heroine";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // Declaring a character constant
   char c1 = '!';
   cout << "The character constant c1 = " << c1 << "." << endl;

   // First member function: concatenates an  object
   // of type basic_string with an object of type basic_string
   string s12 = s1 + s2;
   cout << "The string concatenating s1 & s2 is: " << s12 << endl;

   // Second & fourth member functions: concatenate an object
   // of type basic_string with an object of C-syle string type
   string s1s3 = s1 + s3;
   cout << "The string concatenating s1 & s3 is: " << s1s3 << endl;

   // Third & fifth member functions: concatenate an object
   // of type basic_string with a character constant
   string s1s3c1 = s1s3 + c1;
   cout << "The string concatenating s1 & s3 is: " << s1s3c1 << endl;
}
```

Output

```
The basic_string s1 = anti.
The basic_string s2 = gravity.
The C-style string s3 = heroine.
The character constant c1 = !.
The string concatenating s1 & s2 is: antigravity
```

```
The string concatenating s1 & s3 is: antiheroine
The string concatenating s1 & s3 is: antiheroine!
```

operator!=

Tests if the string object on the left side of the operator is not equal to the string object on the right side.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator!=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator!=(
    const basic_string<CharType, Traits, Allocator>& left,
const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator!=(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is not lexicographically equal to the string object on the right side; otherwise **`false`**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of

characters and their respective character values are the same. Otherwise, they are unequal.

Example

C++

```
// string_op_ne.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "pluck" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "pluck";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 != s2 )
      cout << "The strings s1 & s2 are not equal." << endl;
   else
      cout << "The strings s1 & s2 are equal." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s1 != s3 )
      cout << "The strings s1 & s3 are not equal." << endl;
   else
      cout << "The strings s1 & s3 are equal." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s3 != s2 )
      cout << "The strings s3 & s2 are not equal." << endl;
   else
      cout << "The strings s3 & s2 are equal." << endl;
}
```

Output

```
The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.
```

operator==

Tests if the string object on the left side of the operator is equal to the string object on the right side.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator==(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator==(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator==(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is lexicographically equal to the string object on the right side; otherwise **`false`**.

Remarks

The comparison between string objects is based on a pairwise lexicographical comparison of their characters. Two strings are equal if they have the same number of characters and their respective character values are the same. Otherwise, they are unequal.

Example

C++

```
// string_op_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "pluck" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "pluck";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 == s2 )
      cout << "The strings s1 & s2 are equal." << endl;
   else
      cout << "The strings s1 & s2 are not equal." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s1 == s3 )
      cout << "The strings s1 & s3 are equal." << endl;
   else
      cout << "The strings s1 & s3 are not equal." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s3 == s2 )
      cout << "The strings s3 & s2 are equal." << endl;
   else
      cout << "The strings s3 & s2 are not equal." << endl;
}
```

Output

```
The basic_string s1 = pluck.
The basic_string s2 = strum.
The C-style string s3 = pluck.
The strings s1 & s2 are not equal.
The strings s1 & s3 are equal.
The strings s3 & s2 are not equal.
```

`operator<`

Tests if the string object on the left side of the operator is less than to the string object on the right side.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator<(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator<(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator<(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is lexicographically less than the string object on the right side; otherwise **`false`**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

C++

```
// string_op_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "strict";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 < s2 )
      cout << "The string s1 is less than the string s2." << endl;
   else
      cout << "The string s1 is not less than the string s2." << endl;

   // Second member function: comparison between left-hand object
   // of type basic_string & right-hand object of C-syle string type
   if ( s1 < s3 )
      cout << "The string s1 is less than the string s3." << endl;
   else
      cout << "The string s1 is not less than the string s3." << endl;
```

```
   // Third member function: comparison between left-hand object
   // of C-syle string type & right-hand object of type basic_string
   if ( s3 < s2 )
      cout << "The string s3 is less than the string s2." << endl;
   else
      cout << "The string s3 is not less than the string s2." << endl;
}
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than the string s2.
The string s1 is not less than the string s3.
The string s3 is less than the string s2.
```

`operator<=`

Tests if the string object on the left side of the operator is less than or equal to the string object on the right side.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator<=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator<=(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator<=(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is lexicographically less than or equal to the string object on the right side; otherwise `false`.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, so the strings are equal.

Example

C++

```
// string_op_le.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "strict";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 <= s2 )
      cout << "The string s1 is less than or equal to "
           << "the string s2." << endl;
   else
```

```
      cout << "The string s1 is greater than "
           << "the string s2." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s1 <= s3 )
      cout << "The string s1 is less than or equal to "
           << "the string s3." << endl;
   else
      cout << "The string s1 is greater than "
           << "the string s3." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type  & right-side object of type basic_string
   if ( s2 <= s3 )
      cout << "The string s2 is less than or equal to "
           << "the string s3." << endl;
   else
      cout << "The string s2 is greater than "
           << "the string s3." << endl;
}
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = strict.
The string s1 is less than or equal to the string s2.
The string s1 is less than or equal to the string s3.
The string s2 is greater than the string s3.
```

operator<<

A template function that writes a string into the output stream.

C++

```
template <class CharType, class Traits, class Allocator>
basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& _Ostr,
    const basic_string<CharType, Traits, Allocator>& str);
```

Parameters

_Ostr

The output stream being written to.

str
The string to be entered into the output stream.

Return Value

Writes the value of the specified string to the output stream *_Ostr*.

Remarks

The template function overloads **operator**<< to insert an object *str* of class template basic_string into the stream *_Ostr*. The function effectively returns `_Ostr.write( str.c_str, str.size )`.

`operator>`

Tests if the string object on the left side of the operator is greater than to the string object on the right side.

```
C++
```

```
template <class CharType, class Traits, class Allocator>
bool operator>(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator>(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator>(
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is lexicographically greater than the string object on the right side; otherwise `false`.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds that the strings have the same number of characters, and so the strings are equal.

Example

C++

```
// string_op_gt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "stricture";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 > s2 )
      cout << "The string s1 is greater than "
           << "the string s2." << endl;
   else
```

```
      cout << "The string s1 is not greater than "
           << "the string s2." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s3 > s1 )
      cout << "The string s3 is greater than "
           << "the string s1." << endl;
   else
      cout << "The string s3 is not greater than "
           << "the string s1." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s2 > s3 )
      cout << "The string s2 is greater than "
           << "the string s3." << endl;
   else
      cout << "The string s2 is not greater than "
           << "the string s3." << endl;
}
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is not greater than the string s2.
The string s3 is greater than the string s1.
The string s2 is greater than the string s3.
```

operator>=

Tests if the string object on the left side of the operator is greater than or equal to the string object on the right side.

C++

```
template <class CharType, class Traits, class Allocator>
bool operator>=(
    const basic_string<CharType, Traits, Allocator>& left,
    const basic_string<CharType, Traits, Allocator>& right);

template <class CharType, class Traits, class Allocator>
bool operator>=(
    const basic_string<CharType, Traits, Allocator>& left,
    const CharType* right);

template <class CharType, class Traits, class Allocator>
bool operator>=(
```

```
    const CharType* left,
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

left
A C-style string or an object of type `basic_string` to be compared.

right
A C-style string or an object of type `basic_string` to be compared.

Return Value

`true` if the string object on the left side of the operator is lexicographically greater than or equal to the string object on the right side; otherwise **`false`**.

Remarks

A lexicographical comparison between strings compares them character by character until:

- It finds two corresponding characters unequal, and the result of their comparison is taken as the result of the comparison between the strings.
- It finds no inequalities, but one string has more characters than the other, and the shorter string is considered less than the longer string.
- It finds no inequalities and finds the strings have the same number of characters, and so the strings are equal.

Example

C++

```
// string_op_ge.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
```

```
   string s1 ( "strict" );
   string s2 ( "strum" );
   cout << "The basic_string s1 = " << s1 << "." << endl;
   cout << "The basic_string s2 = " << s2 << "." << endl;

   // Declaring a C-style string
   char *s3 = "stricture";
   cout << "The C-style string s3 = " << s3 << "." << endl;

   // First member function: comparison between left-side object
   // of type basic_string & right-side object of type basic_string
   if ( s1 >= s2 )
      cout << "The string s1 is greater than or equal to "
           << "the string s2." << endl;
   else
      cout << "The string s1 is less than "
           << "the string s2." << endl;

   // Second member function: comparison between left-side object
   // of type basic_string & right-side object of C-syle string type
   if ( s3 >= s1 )
      cout << "The string s3 is greater than or equal to "
           << "the string s1." << endl;
   else
      cout << "The string s3 is less than "
           << "the string s1." << endl;

   // Third member function: comparison between left-side object
   // of C-syle string type & right-side object of type basic_string
   if ( s2 >= s3 )
      cout << "The string s2 is greater than or equal to "
           << "the string s3." << endl;
   else
      cout << "The string s2 is less than "
           << "the string s3." << endl;
}
```

Output

```
The basic_string s1 = strict.
The basic_string s2 = strum.
The C-style string s3 = stricture.
The string s1 is less than the string s2.
The string s3 is greater than or equal to the string s1.
The string s2 is greater than or equal to the string s3.
```

operator>>

A template function that reads a string from an input stream.

```
```

C++

```
template <class CharType, class Traits, class Allocator>
basic_istream<CharType, Traits>& operator>>(
    basic_istream<CharType, Traits>& _Istr,
    basic_string<CharType, Traits, Allocator>& right);
```

Parameters

_Istr
The input stream used to extract the sequence

right
The string that is being extracted from the input stream.

Return Value

Reads the value of the specified string from *_Istr* and returns it into *right*.

Remarks

The operator skips the leading white spaces unless the `skipws` flag is set. It reads all the following characters until the next character is a white space or the end of the file is reached.

The template function overloads **operator>>** to replace the sequence controlled by *right* with a sequence of elements extracted from the stream *_Istr*. Extraction stops:

- At end of file.
- After the function extracts `_Istr`. **width** elements, if that value is nonzero.

After the function extracts `_Istr`. max_size elements.

- After the function extracts an element *ch* for which use_facet< **ctype**< **CharType**> >(`getloc`). **is**(**ctype**< **CharType**>:: **space**, *ch*) is true, in which case the character is put back.

If the function extracts no elements, it calls setstate(`ios_base::failbit`). In any case, it calls **istr**. **width**(0) and returns * `this`.

Example

C++

```
// string_op_read_.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string c0;
   cout << "Input a string c0 ( try: Fibonacci numbers ): ";
   cin >> c0;
   cout << "The string entered is c0 = " << c0 << endl;
}
```

See also

<string>

`<string>` typedefs

Article • 12/06/2021

string
u16string
u32string
wstring

`string`

A type that describes a specialization of the class template basic_string with elements of type `char`.

Other typedefs that specialize `basic_string` include wstring, u16string, and u32string.

```
C++
```

```
typedef basic_string<char, char_traits<char>, allocator<char>> string;
```

Remarks

The following are equivalent declarations:

```
C++
```

```
string str("");

basic_string<char> str("");
```

For a list of string constructors, see basic_string::basic_string.

`u16string`

A type that describes a specialization of the class template basic_string with elements of type `char16_t`.

Other typedefs that specialize `basic_string` include wstring, string, and u32string.

```
C++
```

```
typedef basic_string<char16_t, char_traits<char16_t>, allocator<char16_t>>
u16string;
```

Remarks

For a list of string constructors, see basic_string::basic_string.

`u32string`

A type that describes a specialization of the class template basic_string with elements of type `char32_t`.

Other typedefs that specialize `basic_string` include string, u16string, and wstring.

```
C++
```

```
typedef basic_string<char32_t, char_traits<char32_t>, allocator<char32_t>>
u32string;
```

Remarks

For a list of string constructors, see basic_string::basic_string.

`wstring`

A type that describes a specialization of the class template basic_string with elements of type `wchar_t`.

Other typedefs that specialize `basic_string` include string, u16string, and u32string.

```
C++
```

```
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>
wstring;
```

Remarks

The following are equivalent declarations:

```
C++
```

```
wstring wstr(L"");

basic_string<wchar_t> wstr(L"");
```

For a list of string constructors, see basic_string::basic_string.

> ⓘ **Note**
>
> The size of `wchar_t` is implementation-defined. If your code depends on `wchar_t` to be a certain size, check your platform's implementation (for example, with `sizeof(wchar_t)`). If you need a string character type with a width that is guaranteed to remain the same on all platforms, use **string**, **u16string**, or **u32string**.

See also

<string>

`basic_string` Class

Article • 07/11/2023

The sequences controlled by an object of type `basic_string` are the Standard C++ string class and are referred to as strings, but they shouldn't be confused with the null-terminated C-style strings used throughout the C++ Standard Library. The Standard C++ string is a container that enables the use of strings as normal types, such as comparison and concatenation operations, iterators, C++ Standard Library algorithms, and copying and assigning with class allocator-managed memory. If you need to convert a Standard C++ string to a null-terminated C-style string, use the basic_string::c_str member.

Syntax

C++

```
template <class CharType, class Traits = char_traits<CharType>, class
Allocator = allocator<CharType>>
class basic_string;
```

Parameters

`CharType`

The data type of a single character to be stored in the string. The C++ Standard Library provides specializations of this class template, with the type definitions string for elements of type `char`, wstring, for `wchar_t`, u16string for `char16_t`, and u32string for `char32_t`.

`Traits`

Various important properties of the `CharType` elements in a basic_string specialization are described by the class `Traits`. The default value is `char_traits<CharType>`.

`Allocator`

The type that represents the stored allocator object that encapsulates details about the string's allocation and deallocation of memory. The default value is `allocator<CharType>`.

Constructors

Constructor	Description
basic_string	Constructs a string that is empty or initialized by specific characters or that is a copy of all or part of some other string object or C-string.

Typedefs

Type name	Description
allocator_type	A type that represents the `allocator` class for a string object.
const_iterator	A type that provides a random-access iterator that can access and read a `const` element in the string.
const_pointer	A type that provides a pointer to a `const` element in a string.
const_reference	A type that provides a reference to a `const` element stored in a string for reading and performing `const` operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any `const` element in the string.
difference_type	A type that provides the difference between two iterators that refer to elements within the same string.
iterator	A type that provides a random-access iterator that can read or modify any element in a string.
npos	An unsigned integral value initialized to -1 that indicates either "not found" or "all remaining characters" when a search function fails.
pointer	A type that provides a pointer to a character element in a string or character array.
reference	A type that provides a reference to an element stored in a string.
reverse_iterator	A type that provides a random-access iterator that can read or modify an element in a reversed string.
size_type	An unsigned integral type for the number of elements in a string.
traits_type	A type for the character traits of the elements stored in a string.
value_type	A type that represents the type of characters stored in a string.

Member functions

Member function	Description
append	Adds characters to the end of a string.
assign	Assigns new character values to the contents of a string.
at	Returns a reference to the element at a specified location in the string.
back	
begin	Returns an iterator addressing the first element in the string.
c_str	Converts the contents of a string as a C-style, null-terminated, string.
capacity	Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.
cbegin	Returns a const iterator addressing the first element in the string.
cend	Returns a const iterator that addresses the location succeeding the last element in a string.
clear	Erases all elements of a string.
compare	Compares a string with a specified string to determine if the two strings are equal or if one is lexicographically less than the other.
copy	Copies at most a specified number of characters from an indexed position in a source string to a target character array. Deprecated. Use basic_string::_Copy_s instead.
crbegin	Returns a const iterator that addresses the first element in a reversed string.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed string.
_Copy_s	**Microsoft Specific**: Copies at most a specified number of characters from an indexed position in a source string to a target character array.
data	Converts the contents of a string into an array of characters.
empty	Tests whether the string contains characters.
end	Returns an iterator that addresses the location succeeding the last element in a string.
ends_with C++20	Checks whether the string ends with the specified suffix.
erase	Removes an element or a range of elements in a string from a specified position.

Member function	Description
find	Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.
find_first_not_of	Searches through a string for the first character that isn't any element of a specified string.
find_first_of	Searches through a string for the first character that matches any element of a specified string.
find_last_not_of	Searches through a string for the last character that isn't any element of a specified string.
find_last_of	Searches through a string for the last character that is an element of a specified string.
front	Returns a reference to the first element in a string.
get_allocator	Returns a copy of the `allocator` object used to construct the string.
insert	Inserts an element, several elements, or a range of elements into the string at a specified position.
length	Returns the current number of elements in a string.
max_size	Returns the maximum number of characters a string could contain.
pop_back	Erases the last element of the string.
push_back	Adds an element to the end of the string.
rbegin	Returns an iterator to the first element in a reversed string.
rend	Returns an iterator that points just beyond the last element in a reversed string.
replace	Replaces elements in a string at a specified position with specified characters or characters copied from other ranges or strings or C-strings.
reserve	Sets the capacity of the string to a number at least as great as a specified number.
resize	Specifies a new size for a string, appending or erasing elements as required.
rfind	Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.
shrink_to_fit	Discards the excess capacity of the string.
size	Returns the current number of elements in a string.

Member function	Description
starts_with[C++20]	Checks whether the string starts with the specified prefix.
substr	Copies a substring of at most some number of characters from a string beginning from a specified position.
swap	Exchange the contents of two strings.

Operators

Operator	Description
operator+=	Appends characters to a string.
operator=	Assigns new character values to the contents of a string.
operator[]	Provides a reference to the character with a specified index in a string.

Literals

The headers that define `basic_string` also define the following user-defined literals, which create a string of the specified type from the input parameters.

Declaration	Description
`inline string operator"" s(const char* str, size_t len)`	Returns: `string(str, len)`
`inline string operator"" s(const wchar_t* str, size_t len)`	Returns: `wstring(str, len)`
`inline basic_string<char8_t> operator"" s(const char8_t* str, size_t len)`	Returns: `basic_string<char8_t>(str, len)`
`inline u16string operator"" s(const char16_t* str, size_t len)`	Returns: `u16string(str, len)`
`inline u32string operator"" s(const char32_t* str, size_t len)`	Returns: `u32string(str, len)`

Remarks

If a function is asked to generate a sequence longer than max_size elements, the function reports a length error by throwing an object of type length_error.

References, pointers, and iterators that designate elements of the controlled sequence can become invalid after any call to a function that alters the controlled sequence, or after the first call to a non-`const` member function.

Requirements

Header: <string>

Namespace: std

`basic_string::allocator_type`

A type that represents the allocator class for a string object.

C++

```
typedef Allocator allocator_type;
```

Remarks

The type is a synonym for the template parameter `Allocator`.

Example

C++

```
// basic_string_allocator_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   string s1;
   basic_string <char>::allocator_type xchar = s1.get_allocator( );
   // You can now call functions on the allocator class xchar used by s1
}
```

`basic_string::append`

Adds characters to the end of a string.

C++

```
basic_string<CharType, Traits, Allocator>& append(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& append(
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& append(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset,
    size_type count);

basic_string<CharType, Traits, Allocator>& append(
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& append(
    size_type count,
    value_type char_value);

template <class InputIterator>
basic_string<CharType, Traits, Allocator>& append(
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& append(
    const_pointer first,
    const_pointer last);

basic_string<CharType, Traits, Allocator>& append(
    const_iterator first,
    const_iterator last);
```

Parameters

ptr

The C-string to be appended.

str

The string whose characters are to be appended.

offset

The index of the part of the source string supplying the characters to be appended.

count

The number of characters to be appended, at most, from the source string.

char_value

The character value to be appended.

first

An input iterator addressing the first element in the range to be appended.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the position of the one beyond the last element in the range to be appended.

Return value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the operator+= or the member functions `append` or push_back. `operator+=` appends single-argument values while the multiple-argument `append` member function allows a specific part of a string to be specified for adding.

Example

C++

```
// basic_string_append.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // appending a C-string to a string
   string str1a ( "Hello " );
   cout << "The original string str1 is: " << str1a << endl;
   const char *cstr1a = "Out There ";
   cout << "The C-string cstr1a is: " << cstr1a << endl;
   str1a.append ( cstr1a );
   cout << "Appending the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

   // The second member function
```

```
    // appending part of a C-string to a string
    string str1b ( "Hello " );
    cout << "The string str1b is: " << str1b << endl;
    const char *cstr1b = "Out There ";
    cout << "The C-string cstr1b is: " << cstr1b << endl;
    str1b.append ( cstr1b , 3 );
    cout << "Appending the 1st part of the C-string cstr1b "
         << "to string str1 gives: " << str1b << "."
         << endl << endl;

    // The third member function
    // appending part of one string to another
    string str1c ( "Hello " ), str2c ( "Wide World " );
    cout << "The string str2c is: " << str2c << endl;
    str1c.append ( str2c , 5 , 5 );
    cout << "The appended string str1 is: "
         << str1c << "." << endl << endl;

    // The fourth member function
    // appending one string to another in two ways,
    // comparing append and operator [ ]
    string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World " );
    cout << "The  string str2d is: " << str2d << endl;
    str1d.append ( str2d );
    cout << "The appended string str1d is: "
         << str1d << "." << endl;
    str1d += str3d;
    cout << "The doubly appended strig str1 is: "
         << str1d << "." << endl << endl;

    // The fifth member function
    // appending characters to a string
    string str1e ( "Hello " );
    str1e.append ( 4 , '!' );
    cout << "The string str1 appended with exclamations is: "
         << str1e << endl << endl;

    // The sixth member function
    // appending a range of one string to another
    string str1f ( "Hello " ), str2f ( "Wide World " );
    cout << "The string str2f is: " << str2f << endl;
    str1f.append ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
    cout << "The appended string str1 is: "
         << str1f << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello
The C-string cstr1a is: Out There
Appending the C-string cstr1a to string str1 gives: Hello Out There .

The string str1b is: Hello
```

```
The C-string cstr1b is: Out There
Appending the 1st part of the C-string cstr1b to string str1 gives: Hello
Out.

The string str2c is: Wide World
The appended string str1 is: Hello World.

The  string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended strig str1 is: Hello Wide World .

The string str1 appended with exclamations is: Hello !!!!

The string str2f is: Wide World
The appended string str1 is: Hello World.
```

basic_string::assign

Assigns new character values to the contents of a string.

C++

```
basic_string<CharType, Traits, Allocator>& assign(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& assign(
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& assign(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type off,
    size_type count);

basic_string<CharType, Traits, Allocator>& assign(
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& assign(
    size_type count,
    value_type char_value);

template <class InIt>
basic_string<CharType, Traits, Allocator>& assign(
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& assign(
    const_pointer first,
    const_pointer last);

basic_string<CharType, Traits, Allocator>& assign(
```

```
    const_iterator first,
    const_iterator last);
```

Parameters

ptr

A pointer to the characters of the C-string to be assigned to the target string.

count

The number of characters to be assigned, from the source string.

str

The source string whose characters are to be assigned to the target string.

char_value

The character value to be assigned.

first

An input iterator, const_pointer, or const_iterator addressing the first character in the range of the source string to be assigned to the target range.

last

An input iterator, const_pointer, or const_iterator addressing the one beyond the last character in the range of the source string to be assigned to the target range.

off

The position at which new characters will start to be assigned.

Return value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings can be assigned new character values. The new value can be either a string and C-string or a single character. The operator= may be used if the new value can be described by a single parameter; otherwise the member function `assign`, which has multiple parameters, can be used to specify which part of the string is to be assigned to a target string.

Example

C++

```
// basic_string_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function assigning the
   // characters of a C-string to a string
   string str1a;
   const char *cstr1a = "Out There";
   cout << "The C-string cstr1a is: " << cstr1a <<  "." << endl;
   str1a.assign ( cstr1a );
   cout << "Assigning the C-string cstr1a to string str1 gives: "
        << str1a << "." << endl << endl;

   // The second member function assigning a specific
   // number of the of characters a C-string to a string
   string  str1b;
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b << endl;
   str1b.assign ( cstr1b , 3 );
   cout << "Assigning the 1st part of the C-string cstr1b "
        << "to string str1 gives: " << str1b << "."
        << endl << endl;

   // The third member function assigning a specific number
   // of the characters from one string to another string
   string str1c ( "Hello " ), str2c ( "Wide World " );
   cout << "The string str2c is: " << str2c << endl;
   str1c.assign ( str2c , 5 , 5 );
   cout << "The newly assigned string str1 is: "
        << str1c << "." << endl << endl;

   // The fourth member function assigning the characters
   // from one string to another string in two equivalent
   // ways, comparing the assign and operator =
   string str1d ( "Hello" ), str2d ( "Wide" ), str3d ( "World" );
   cout << "The original string str1 is: " << str1d << "." << endl;
   cout << "The string str2d is: " << str2d << endl;
   str1d.assign ( str2d );
   cout << "The string str1 newly assigned with string str2d is: "
        << str1d << "." << endl;
   cout << "The string str3d is: " << str3d << "." << endl;
   str1d = str3d;
   cout << "The string str1 reassigned with string str3d is: "
        << str1d << "." << endl << endl;
```

```
    // The fifth member function assigning a specific
    // number of characters of a certain value to a string
    string str1e ( "Hello " );
    str1e.assign ( 4 , '!' );
    cout << "The string str1 assigned with eclamations is: "
         << str1e << endl << endl;

    // The sixth member function assigning the value from
    // the range of one string to another string
    string str1f ( "Hello " ), str2f ( "Wide World " );
    cout << "The string str2f is: " << str2f << endl;
    str1f.assign ( str2f.begin ( ) + 5 , str2f.end ( ) - 1 );
    cout << "The string str1 assigned a range of string str2f is: "
         << str1f << "." << endl << endl;
}
```

Output

```
The C-string cstr1a is: Out There.
Assigning the C-string cstr1a to string str1 gives: Out There.

The C-string cstr1b is: Out There
Assigning the 1st part of the C-string cstr1b to string str1 gives: Out.

The string str2c is: Wide World
The newly assigned string str1 is: World.

The original string str1 is: Hello.
The string str2d is: Wide
The string str1 newly assigned with string str2d is: Wide.
The string str3d is: World.
The string str1 reassigned with string str3d is: World.

The string str1 assigned with eclamations is: !!!!

The string str2f is: Wide World
The string str1 assigned a range of string str2f is: World.
```

basic_string::at

Provides a reference to the character with a specified index in a string.

C++

```
const_reference at(size_type offset) const;

reference at(size_type offset);
```

Parameters

offset
The index of the position of the element to be referenced.

Return value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero and the following elements are indexed consecutively by the positive integers, so that a string of length *n* has an *n*th element indexed by the number *n* - 1.

The member operator[] is faster than the member function `at` for providing read and write access to the elements of a string.

The member `operator[]` doesn't check whether the index passed as a parameter is valid but the member function `at` does and so should be used if the validity isn't certain. An invalid index, which is an index less that zero or greater than or equal to the size of the string, passed to the member function `at` throws an out_of_range Class exception. An invalid index passed to the `operator[]` results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null-character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non-`const` strings.

Example

C++

```
// basic_string_at.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello world" ), str2 ( "Goodbye world" );
   const string  cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
```

```
   cout << "The original string str1 is: " << str1 << endl;
   cout << "The original string str2 is: " << str2 << endl;

   // Element access to the non const strings
   basic_string <char>::reference refStr1 = str1 [6];
   basic_string <char>::reference refStr2 = str2.at ( 3 );

   cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
   cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

   // Element access to the const strings
   basic_string <char>::const_reference crefStr1 = cstr1 [ cstr1.length ( )
];
   basic_string <char>::const_reference crefStr2 = cstr2.at ( 8 );

   if ( crefStr1 == '\0' )
      cout << "The null character is returned as a valid reference."
           << endl;
   else
      cout << "The null character is not returned." << endl;
   cout << "The character with index 8 in the const string cstr2 is: "
        << crefStr2 << "." << endl;
}
```

`basic_string::back`

Returns a reference to the last element in the string.

C++

```
const_reference back() const;

reference back();
```

Return value

A reference to the last element of the string, which must be non-empty.

Remarks

`basic_string::basic_string`

Constructs a string that is empty, initialized by specific characters, or is a copy of all or part of another string object or C style (null-terminated) string.

C++

```
basic_string();

explicit basic_string(
    const allocator_type& alloc_type);

basic_string(
    const basic_string& right);

basic_string(
    basic_string&& right);

basic_string(
    const basic_string& right,
    size_type right_offset,
    size_type count = npos);

basic_string(
    const basic_string& right,
    size_type right_offset,
    size_type count,
    const allocator_type& alloc_type);

basic_string(
    const value_type* ptr,
    size_type count);

basic_string(
    const value_type* ptr,
    size_type count,
    const allocator_type& alloc_type);

basic_string(
    const value_type* ptr);

basic_string(
    const value_type* ptr,
    const allocator_type& alloc_type);

basic_string(
    size_type count,
    value_type char_value);

basic_string(
    size_type count,
    value_type char_value,
    const allocator_type& alloc_type);

template <class InputIterator>
basic_string(
    InputIterator first,
    InputIterator last);
```

```
template <class InputIterator>
basic_string(
    InputIterator first,
    InputIterator last,
    const allocator_type& alloc_type);

basic_string(
    const_pointer first,
    const_pointer last);

basic_string(
    const_iterator first,
    const_iterator last);
```

Parameters

ptr

The C-string whose characters are to be used to initialize the `string` being constructed. This value can't be a null pointer unless *count* is zero.

alloc_type

The storage allocator class for the string object being constructed.

count

The number of characters to be initialized.

right

The string to initialize the string being constructed.

right_offset

The index of a character in a string that is the first to be used to initialize character values for the string being constructed.

char_value

The character value to be copied into the string being constructed.

first

An input iterator, const_pointer, or const_iterator addressing the first element in the source range to be inserted.

last

An input iterator, const_pointer, or const_iterator addressing the position of the one beyond the last element in the source range to be inserted.

Return value

A reference to the string object that is being constructed by the constructors.

Remarks

All constructors store a basic_string::allocator_type and initialize the controlled sequence. The allocator object is the argument `al`, if present. For the copy constructor, it's `right.get_allocator()`, a call to basic_string::get_allocator. Otherwise, the allocator is `Alloc()`.

The controlled sequence is initialized to a copy of the operand sequence specified by the remaining operands. A constructor without an operand sequence specifies an empty initial controlled sequence. If `InputIterator` is an integer type in a template constructor, the operand sequence `first, last` behaves the same as `(size_type) first, (value_type) last`.

Example

C++

```
// basic_string_ctor.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function initializing with a C-string
   const char *cstr1a = "Hello Out There.";
   basic_string <char> str1a ( cstr1a , 5);
   cout << "The string initialized by C-string cstr1a is: "
        << str1a << "." << endl;

   // The second member function initializing with a string
   string  str2a ( "How Do You Do" );
   basic_string <char> str2b ( str2a , 7 , 7 );
   cout << "The string initialized by part of the string cstr2a is: "
        << str2b << "." << endl;

   // The third member function initializing a string
   // with a number of characters of a specific value
   basic_string <char> str3a ( 5, '9' );
   cout << "The string initialized by five number 9s is: "
        << str3a << endl;

   // The fourth member function creates an empty string
   // and string with a specified allocator
```

```
   basic_string <char> str4a;
   string str4b;
   basic_string <char> str4c ( str4b.get_allocator( ) );
   if (str4c.empty ( ) )
      cout << "The string str4c is empty." << endl;
   else
      cout << "The string str4c is not empty." << endl;

   // The fifth member function initializes a string from
   // another range of characters
   string str5a ( "Hello World" );
   basic_string <char> str5b ( str5a.begin ( ) + 5 , str5a.end ( ) );
   cout << "The string initialized by another range is: "
        << str5b << "." << endl;
}
```

basic_string::begin

Returns an iterator addressing the first element in the string.

C++

```
const_iterator begin() const;

iterator begin();
```

Return value

A random-access iterator that addresses the first element of the sequence or just beyond the end of an empty sequence.

Example

C++

```
// basic_string_begin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
   using namespace std;
   string str1 ( "No way out." ), str2;
   basic_string <char>::iterator strp_Iter, str1_Iter, str2_Iter;
   basic_string <char>::const_iterator str1_cIter;

   str1_Iter = str1.begin ( );
```

```
   cout << "The first character of the string str1 is: "
        << *str1_Iter << endl;
   cout << "The full original string str1 is: " << str1 << endl;

   // The dereferenced iterator can be used to modify a character
*str1_Iter = 'G';
   cout << "The first character of the modified str1 is now: "
        << *str1_Iter << endl;
   cout << "The full modified string str1 is now: " << str1 << endl;

   // The following line would be an error because iterator is const
   // *str1_cIter = 'g';

   // For an empty string, begin is equivalent to end
   if (  str2.begin ( ) == str2.end ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The string str2 is not empty." << endl;
}
```

`basic_string::c_str`

Converts the contents of a string as a C-style, null-terminated string.

C++

```
const value_type *c_str() const;
```

Return value

A pointer to the C-style version of the invoking string. The pointer value isn't valid after calling a non-`const` function, including the destructor, in the `basic_string` class on the object.

Remarks

Objects of type string belonging to the class template `basic_string<char>` aren't necessarily null terminated. The null character `'\0'` is used as a special character in a C-string to mark the end of the string but has no special meaning in an object of type string and may be a part of the string just like any other character. There's an automatic conversion from `const char *` into strings, but the string class doesn't provide for automatic conversions from C-style strings to objects of type `basic_string<char>`.

The returned C-style string shouldn't be modified, which could invalidate the pointer to the string, or deleted, as the string has a limited lifetime and is owned by the class string.

Example

C++

```
// basic_string_c_str.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string  str1 ( "Hello world" );
   cout << "The original string object str1 is: "
        << str1 << endl;
   cout << "The length of the string object str1 = "
        << str1.length ( ) << endl << endl;

   // Converting a string to an array of characters
   const char *ptr1 = 0;
   ptr1= str1.data ( );
   cout << "The modified string object ptr1 is: " << ptr1
        << endl;
   cout << "The length of character array str1 = "
        << strlen ( ptr1) << endl << endl;

   // Converting a string to a C-style string
   const char *c_str1 = str1.c_str ( );
   cout << "The C-style string c_str1 is: " << c_str1
        << endl;
   cout << "The length of C-style string str1 = "
        << strlen ( c_str1) << endl << endl;
}
```

Output

```
The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11

The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11
```

`basic_string::capacity`

Returns the largest number of elements that could be stored in a string without increasing the memory allocation of the string.

C++

```
size_type capacity() const;
```

Return value

The size of storage currently allocated in memory to hold the string.

Remarks

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as size.

Example

C++

```
// basic_string_capacity.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
```

```
    cout << "The capacity of original string str1 is: "
         << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
         << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size (  );
    lenStr1 = str1.length (  );
    capStr1 = str1.capacity (  );
    max_sizeStr1 = str1.max_size (  );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
         << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
         << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
         << max_sizeStr1 << "." << endl;
}
```

`basic_string::cbegin`

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used along with the

auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-`const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++

auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`basic_string::cend`

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++

const_iterator cend() const;
```

Return value

A `const` random-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used along with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non-`const`) container of any kind that supports `end()` and `cend()`.

```
C++

auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`basic_string::clear`

Erases all elements of a string.

C++

```
void clear();
```

Remarks

The string on which the member function is called will be empty.

Example

C++

```
// basic_string_clear.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ("Hello world"), str2;
   basic_string <char>::iterator str_Iter;
   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   str1.clear ( );
   cout << "The modified string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   //For an empty string, begin is equivalent to end
   if ( str1.begin ( ) == str1.end ( ) )
      cout << "Nothing printed above because "
           << "the string str1 is empty." << endl;
   else
      cout << "The string str1 is not empty." << endl;
}
```

Output

```
The original string str1 is: Hello world
The modified string str1 is:
Nothing printed above because the string str1 is empty.
```

`basic_string::compare`

Does a case-sensitive comparison with a specified string to determine if the two strings are equal or if one is lexicographically less than the other.

C++

```
int compare(
    const basic_string<CharType, Traits, Allocator>& str) const;

int compare(
    size_type position_1,
    size_type number_1,
    const basic_string<CharType, Traits, Allocator>& str) const;

int compare(
    size_type position_1,
    size_type number_1,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset,
    size_type count) const;

int compare(
    const value_type* ptr) const;

int compare(
    size_type position_1,
    size_type number_1,
    const value_type* ptr) const;

int compare(
    size_type position_1,
    size_type number_1,
    const value_type* ptr
    size_type number_2) const;
```

Parameters

`str`

The string that is to be compared to the operand string.

position_1

The index of the operand string at which the comparison begins.

number_1

The maximum number of characters from the operand string to be compared.

number_2

The maximum number of characters from the parameter string to be compared.

offset

The index of the parameter string at which the comparison begins.

count

The maximum number of characters from the parameter string to be compared.

ptr

The C-string to be compared to the operand string.

Return value

A negative value if the operand string is less than the parameter string; zero if the two strings are equal; or a positive value if the operand string is greater than the parameter string.

Remarks

The `compare` member functions compare either all, or part, of the parameter and operand strings depending on which in used.

The comparison is case-sensitive.

Example

C++

```
// basic_string_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function compares
```

```
    // an operand string to a parameter string
    int comp1;
    string s1o ( "CAB" );
    string s1p ( "CAB" );
    cout << "The operand string is: " << s1o << endl;
    cout << "The parameter string is: " << s1p << endl;
    comp1 = s1o.compare ( s1p );
    if ( comp1 < 0 )
       cout << "The operand string is less than "
            << "the parameter string." << endl;
    else if ( comp1 == 0 )
       cout << "The operand string is equal to "
            << "the parameter string." << endl;
    else
       cout << "The operand string is greater than "
            << "the parameter string." << endl;
    cout << endl;

    // The second member function compares part of
    // an operand string to a parameter string
    int comp2a, comp2b;
    string s2o ( "AACAB" );
    string s2p ( "CAB" );
    cout << "The operand string is: " << s2o << endl;
    cout << "The parameter string is: " << s2p << endl;
    comp2a = s2o.compare (  2 , 3 , s2p );
    if ( comp2a < 0 )
       cout << "The last three characters of "
            << "the operand string\n are less than "
            << "the parameter string." << endl;
    else if ( comp2a == 0 )
       cout << "The last three characters of "
            << "the operand string\n are equal to "
            << "the parameter string." << endl;
    else
       cout << "The last three characters of "
            << "the operand string\n is greater than "
            << "the parameter string." << endl;

    comp2b = s2o.compare (  0 , 3 , s2p );
    if ( comp2b < 0 )
       cout << "The first three characters of "
            << "the operand string\n are less than "
            << "the parameter string." << endl;
    else if ( comp2b == 0 )
       cout << "The first three characters of "
            << "the operand string\n are equal to "
            << "the parameter string." << endl;
    else
       cout << "The first three characters of "
            << "the operand string\n is greater than "
            << "the parameter string." << endl;
    cout << endl;

    // The third member function compares part of
```

```
    // an operand string to part of a parameter string
    int comp3a;
    string s3o ( "AACAB" );
    string s3p ( "DCABD" );
    cout << "The operand string is: " << s3o << endl;
    cout << "The parameter string is: " << s3p << endl;
    comp3a = s3o.compare (  2 , 3 , s3p , 1 , 3 );
    if ( comp3a < 0 )
       cout << "The three characters from position 2 of "
            << "the operand string are less than\n "
            << "the 3 characters parameter string "
            << "from position 1." << endl;
    else if ( comp3a == 0 )
       cout << "The three characters from position 2 of "
            << "the operand string are equal to\n "
            << "the 3 characters parameter string "
            << "from position 1." << endl;
    else
       cout << "The three characters from position 2 of "
            << "the operand string is greater than\n "
            << "the 3 characters parameter string "
            << "from position 1." << endl;
    cout << endl;

    // The fourth member function compares
    // an operand string to a parameter C-string
    int comp4a;
    string s4o ( "ABC" );
    const char* cs4p = "DEF";
    cout << "The operand string is: " << s4o << endl;
    cout << "The parameter C-string is: " << cs4p << endl;
    comp4a = s4o.compare ( cs4p );
    if ( comp4a < 0 )
       cout << "The operand string is less than "
            << "the parameter C-string." << endl;
    else if ( comp4a == 0 )
       cout << "The operand string is equal to "
            << "the parameter C-string." << endl;
    else
       cout << "The operand string is greater than "
            << "the parameter C-string." << endl;
    cout << endl;

    // The fifth member function compares part of
    // an operand string to a parameter C-string
    int comp5a;
    string s5o ( "AACAB" );
    const char* cs5p = "CAB";
    cout << "The operand string is: " << s5o << endl;
    cout << "The parameter string is: " << cs5p << endl;
    comp5a = s5o.compare (  2 , 3 , s2p );
    if ( comp5a < 0 )
       cout << "The last three characters of "
            << "the operand string\n are less than "
            << "the parameter C-string." << endl;
```

```
   else if ( comp5a == 0 )
      cout << "The last three characters of "
           << "the operand string\n are equal to "
           << "the parameter C-string." << endl;
   else
      cout << "The last three characters of "
           << "the operand string\n is greater than "
           << "the parameter C-string." << endl;
   cout << endl;

   // The sixth member function compares part of
   // an operand string to part of an equal length of
   // a parameter C-string
   int comp6a;
   string s6o ( "AACAB" );
   const char* cs6p = "ACAB";
   cout << "The operand string is: " << s6o << endl;
   cout << "The parameter C-string is: " << cs6p << endl;
   comp6a = s6o.compare (  1 , 3 , cs6p , 3 );
   if ( comp6a < 0 )
      cout << "The 3 characters from position 1 of "
           << "the operand string are less than\n "
           << "the first 3 characters of the parameter C-string."
           << endl;
   else if ( comp6a == 0 )
      cout << "The 3 characters from position 2 of "
           << "the operand string are equal to\n "
           << "the first 3 characters of the parameter C-string."
           <<  endl;
   else
      cout << "The 3 characters from position 2 of "
           << "the operand string is greater than\n "
           << "the first 3 characters of the parameter C-string."
           << endl;
   cout << endl;
}
```

Output

```
The operand string is: CAB
The parameter string is: CAB
The operand string is equal to the parameter string.

The operand string is: AACAB
The parameter string is: CAB
The last three characters of the operand string
are equal to the parameter string.
The first three characters of the operand string
are less than the parameter string.

The operand string is: AACAB
The parameter string is: DCABD
The three characters from position 2 of the operand string are equal to
```

```
the 3 characters parameter string from position 1.

The operand string is: ABC
The parameter C-string is: DEF
The operand string is less than the parameter C-string.

The operand string is: AACAB
The parameter string is: CAB
The last three characters of the operand string
are equal to the parameter C-string.

The operand string is: AACAB
The parameter C-string is: ACAB
The 3 characters from position 2 of the operand string are equal to
the first 3 characters of the parameter C-string.
```

`basic_string::const_iterator`

A type that provides a random-access iterator that can access and read a `const` element in the string.

C++

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for begin for an example of how to declare and use `const_iterator`.

`basic_string::const_pointer`

A type that provides a pointer to a `const` element in a string.

C++

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

The type is a synonym for `allocator_type::const_pointer`.

For type `string`, it's equivalent to `char*`.

Pointers that are declared const must be initialized when they're declared. Const pointers always point to the same memory location and may point to constant or non-constant data.

Example

C++

```
// basic_string_const_ptr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   basic_string<char>::const_pointer pstr1a = "In Here";
   const char *cstr1c = "Out There";

   cout << "The string pstr1a is: " << pstr1a <<  "." << endl;
   cout << "The C-string cstr1c is: " << cstr1c << "." << endl;
}
```

Output

```
The string pstr1a is: In Here.
The C-string cstr1c is: Out There.
```

`basic_string::const_reference`

A type that provides a reference to a `const` element stored in a string for reading and performing `const` operations.

C++

```
typedef typename allocator_type::const_reference const_reference;
```

Remarks

A type `const_reference` can't be used to modify the value of an element.

The type is a synonym for `allocator_type::const_reference`. For type `string`, it's equivalent to const `char&`.

Example

See the example for at for an example of how to declare and use `const_reference`.

`basic_string::const_reverse_iterator`

A type that provides a random-access iterator that can read any `const` element in the string.

```
C++
```

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for rbegin for an example of how to declare and use `const_reverse_iterator`.

`basic_string::copy`

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using basic_string::_Copy_s instead.

```
C++
```

```
size_type copy(
    value_type* ptr,
    size_type count,
    size_type offset = 0) const;
```

Parameters

ptr

The target character array to which the elements are to be copied.

count The number of characters to be copied, at most, from the source string.

offset

The beginning position in the source string from which copies are to be made.

Return value

The number of characters copied.

Remarks

A null character isn't appended to the end of the copy.

Example

C++

```
// basic_string_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello World" );
   basic_string <char>::iterator str_Iter;
   char array1 [ 20 ] = { 0 };
   char array2 [ 10 ] = { 0 };
   basic_string <char>:: pointer array1Ptr = array1;
   basic_string <char>:: value_type *array2Ptr = array2;

   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
```

```
    cout << endl;

    basic_string <char>:: size_type nArray1;
    // Note: string::copy is potentially unsafe, consider
    // using string::_Copy_s instead.
    nArray1 = str1.copy ( array1Ptr , 12 );  // C4996
    cout << "The number of copied characters in array1 is: "
         << nArray1 << endl;
    cout << "The copied characters array1 is: " << array1 << endl;

    basic_string <char>:: size_type nArray2;
    // Note: string::copy is potentially unsafe, consider
    // using string::_Copy_s instead.
    nArray2 = str1.copy ( array2Ptr , 5 , 6  );  // C4996
    cout << "The number of copied characters in array2 is: "
            << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
}
```

Output

```
The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
The number of copied characters in array2 is: 5
The copied characters array2 is: World
```

`basic_string::crbegin`

Returns a const iterator that addresses the first element in a reversed string.

C++

```
const_reverse_iterator crbegin() const;
```

Return value

A reverse iterator that points just beyond the end of the string. The position designates the beginning of the reverse string.

`basic_string::crend`

Returns a `const` iterator that addresses the location succeeding the last element in a reversed string.

```
C++
```

```
const_reverse_iterator crend() const;
```

Return value

A `const` reverse iterator that addresses the location succeeding the last element in a reversed string (the location that had preceded the first element in the unreversed string).

Remarks

`basic_string::_Copy_s`

Copies at most a specified number of characters from an indexed position in a source string to a target character array.

```
C++
```

```
size_type _Copy_s(
    value_type* dest,
    size_type dest_size,
    size_type count,
    size_type offset = 0) const;
```

Parameters

`dest`
The target character array to which the elements are to be copied.

`dest_size`
The size of *dest*.

`count` The number of characters to be copied, at most, from the source string.

`offset`
The beginning position in the source string from which copies are to be made.

Return value

The number of characters copied.

Remarks

A null character isn't appended to the end of the copy. This function is Microsoft specific.

Example

C++

```
// basic_string__Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;
    string str1("Hello World");
    basic_string<char>::iterator str_Iter;
    const int array1_size = 20;
    char array1[array1_size] = { 0 };
    const int array2_size = 10;
    char array2[array2_size] = { 0 };
    basic_string<char>:: pointer array1Ptr = array1;
    basic_string<char>:: value_type *array2Ptr = array2;

    cout << "The original string str1 is: ";
    for (str_Iter = str1.begin(); str_Iter != str1.end(); str_Iter++)
        cout << *str_Iter;
    cout << endl;

    basic_string<char>::size_type nArray1;
    nArray1 = str1._Copy_s(array1Ptr, array1_size, 12);
    cout << "The number of copied characters in array1 is: "
         << nArray1 << endl;
    cout << "The copied characters array1 is: " << array1 << endl;

    basic_string<char>:: size_type nArray2;
    nArray2 = str1._Copy_s(array2Ptr, array2_size, 5, 6);
    cout << "The number of copied characters in array2 is: "
         << nArray2 << endl;
    cout << "The copied characters array2 is: " << array2Ptr << endl;
}
```

Output

```
The original string str1 is: Hello World
The number of copied characters in array1 is: 11
The copied characters array1 is: Hello World
```

```
The number of copied characters in array2 is: 5
The copied characters array2 is: World
```

`basic_string::data`

Converts the contents of a string into a null-terminated array of characters.

C++

```
const value_type *data() const noexcept;
value_type *data() noexcept;
```

Return value

A pointer to the first element of the null-terminated array containing the contents of the string. For an empty string, the pointer points to a single null character equal to `value_type()`.

Remarks

The pointer returned by `data` points at a valid range `[data(), data() + size()]`. Each element in the range corresponds to the current data in the string. That is, for every valid offset *`n`* in the range, `data() + n == addressof(operator[](n))`.

If you modify the contents of the string returned by the `const` overload of `data`, the behavior is undefined. You also get undefined behavior if the terminal null character is changed to any other value. The returned pointer may be invalidated if a non-`const` reference to the string is passed to a standard library function. It can also be invalidated by a call to a non-`const` member function. Calls to members `at`, `back`, `begin`, `end`, `front`, `rbegin`, `rend`, and `operator[]` don't invalidate the pointer.

Before C++11, `data` didn't guarantee the returned string was null-terminated. Since C++11, `data` and `c_str` both return a null-terminated string, and are effectively the same.

The non-`const` overload is new in C++17. To use it, specify the **`/std:c++17`** or later compiler option.

Example

C++

```
// basic_string_data.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string str1 ( "Hello world" );
   cout << "The original string object str1 is: "
        << str1 << endl;
   cout << "The length of the string object str1 = "
        << str1.length ( ) << endl << endl;

   // Converting a string to an array of characters
   const char *ptr1 = 0;
   ptr1= str1.data ( );
   cout << "The modified string object ptr1 is: " << ptr1
        << endl;
   cout << "The length of character array str1 = "
        << strlen ( ptr1) << endl << endl;

   // Converting a string to a C-style string
   const char *c_str1 = str1.c_str ( );
   cout << "The C-style string c_str1 is: " << c_str1
        << endl;
   cout << "The length of C-style string str1 = "
        << strlen ( c_str1) << endl << endl;
}
```

Output

```
The original string object str1 is: Hello world
The length of the string object str1 = 11

The modified string object ptr1 is: Hello world
The length of character array str1 = 11

The C-style string c_str1 is: Hello world
The length of C-style string str1 = 11
```

basic_string::difference_type

A type that provides the difference between two iterators that refer to elements within the same string.

C++

```
typedef typename allocator_type::difference_type difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

For type `string`, it's equivalent to `ptrdiff_t`.

Example

C++

```
// basic_string_diff_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "quintillion" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexChFi, indexChLi;

   indexChFi = str1.find_first_of ( "i" );
   indexChLi = str1.find_last_of ( "i" );
   basic_string<char>::difference_type diffi = indexChLi - indexChFi;

   cout << "The first character i is at position: "
        << indexChFi << "." << endl;
   cout << "The last character i is at position: "
        << indexChLi << "." << endl;
   cout << "The difference is: " << diffi << "." << endl;
}
```

Output

```
The original string str1 is: quintillion
The first character i is at position: 2.
The last character i is at position: 8.
The difference is: 6.
```

`basic_string::empty`

Tests whether the string contains characters or not.

C++

```
bool empty() const;
```

Return value

`true` if the string object contains no characters; `false` if it has at least one character.

Remarks

The member function is equivalent to size == 0.

Example

C++

```
// basic_string_empty.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main() {
   using namespace std;

   bool b1, b2;

   string str1 ("Hello world");
   cout << "The original string object str1 is: " << str1 << endl;
   b1 = str1.empty();
   if (b1)
      cout << "The string object str1 is empty." << endl;
   else
      cout << "The string object str1 is not empty." << endl;
   cout << endl;

   // An example of an empty string object
   string str2;
   b2 = str2.empty();
   if (b2)
      cout << "The string object str2 is empty." << endl;
   else
```

```
        cout << "The string object str2 is not empty." << endl;
}
```

basic_string::end

Returns an iterator that addresses the location succeeding the last element in a string.

C++

```
const_iterator end() const;

iterator end();
```

Return value

Returns a random-access iterator that addresses the location succeeding the last element in a string.

Remarks

`end` is often used to test whether an iterator has reached the end of its string. The value returned by `end` shouldn't be dereferenced.

If the return value of `end` is assigned to a `const_iterator`, the string object can't be modified. If the return value of `end` is assigned to an `iterator`, the string object can be modified.

Example

C++

```
// basic_string_end.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "No way out." ), str2;
   basic_string <char>::iterator str_Iter, str1_Iter, str2_Iter;
   basic_string <char>::const_iterator str1_cIter;

   str1_Iter = str1.end ( );
```

```
   str1_Iter--;
   str1_Iter--;
   cout << "The last character-letter of the string str1 is: " << *str1_Iter
<< endl;
   cout << "The full original string str1 is: " << str1 << endl;

   // end used to test when an iterator has reached the end of its string
   cout << "The string is now: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
   *str1_Iter = 'T';
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;
   cout << "The modified string str1 is now: " << str1 << endl;

   // The following line would be an error because iterator is const
   // *str1_cIter = 'T';

   // For an empty string, end is equivalent to begin
   if ( str2.begin( ) == str2.end ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The last character-letter of the string str1 is: t
The full original string str1 is: No way out.
The string is now: No way out.
The last character-letter of the modified str1 is now: T
The modified string str1 is now: No way ouT.
The string str2 is empty.
```

basic_string::ends_with

Check whether the string ends with the specified suffix.

C++

```
bool ends_with(const CharType c) const noexcept;
bool ends_with(const CharType* const x) const noexcept;
bool ends_with(const basic_string_view sv) const noexcept;
```

Parameters

`c`

The single character suffix to look for.

`sv`

A string view containing the suffix to look for.
You can pass a `std::basic_string`, which converts to a string view.

`x`

Null-terminated character string containing the suffix to look for.

Return value

`true` if the string ends with the specified suffix; `false` otherwise.

Remarks

`ends_with()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

See starts_with to check if a string starts with the specified prefix.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <string>
#include <iostream>

int main()
{
    std::basic_string<char> str = "abcdefg";

    std::cout << std::boolalpha; // so booleans show as 'true'/'false'
    std::cout << str.ends_with('g') << '\n';
    std::cout << str.ends_with("eFg") << '\n';

    std::basic_string<char> str2 = "efg";
    std::cout << str.ends_with(str2);

    return 0;
}
```

Output

```
true
false
```

```
true
```

`basic_string::erase`

Removes an element or a range of elements in a string from a specified position.

```
C++
```

```
iterator erase(
    iterator first,
    iterator last);

iterator erase(
    iterator iter);

basic_string<CharType, Traits, Allocator>& erase(
    size_type offset = 0,
    size_type count = npos);
```

Parameters

`first`
An iterator addressing the position of the first element in the range to be erased.

`last`
An iterator addressing the position one past the last element in the range to be erased.

`iter`
An iterator addressing the position of the element in the string to be erased.

`offset`
The index of the first character in the string to be removed.

`count`
The number of elements that will be removed if there are as many in the range of the string beginning with *`offset`*.

Return value

For the first two member functions, an iterator addressing the first character after the last character removed by the member function. For the third member function, a reference to the string object from which the elements have been erased.

Remarks

The third member function returns `*this`.

Example

C++

```
// basic_string_erase.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The 1st member function using a range demarcated
   // by iterators
   string str1 ( "Hello world" );
   basic_string <char>::iterator str1_Iter;
   cout << "The original string object str1 is: "
        << str1 << "." << endl;
   str1_Iter = str1.erase ( str1.begin ( ) + 3 , str1.end ( ) - 1 );
   cout << "The first element after those removed is: "
        << *str1_Iter << "." << endl;
   cout << "The modified string object str1 is: " << str1
           << "." << endl << endl;

   // The 2nd member function erasing a char pointed to
   // by an iterator
   string str2 ( "Hello World" );
   basic_string <char>::iterator str2_Iter;
   cout << "The original string object str2 is: " << str2
        << "." << endl;
   str2_Iter = str2.erase ( str2.begin ( ) + 5 );
   cout << "The first element after those removed is: "
        << *str2_Iter << "." << endl;
   cout << "The modified string object str2 is: " << str2
        << "." << endl << endl;

   // The 3rd member function erasing a number of chars
   // after a char
   string str3 ( "Hello computer" ), str3m;
   basic_string <char>::iterator str3_Iter;
   cout << "The original string object str3 is: "
        << str3 << "." << endl;
   str3m = str3.erase ( 6 , 8 );
   cout << "The modified string object str3m is: "
        << str3m << "." << endl;
}
```

Output

```
The original string object str1 is: Hello world.
The first element after those removed is: d.
The modified string object str1 is: Held.

The original string object str2 is: Hello World.
The first element after those removed is: W.
The modified string object str2 is: HelloWorld.

The original string object str3 is: Hello computer.
The modified string object str3m is: Hello .
```

`basic_string::find`

Searches a string in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.

C++

```
size_type find(
    value_type char_value,
    size_type offset = 0) const;

size_type find(
    const value_type* ptr,
    size_type offset = 0) const;

size_type find(
    const value_type* ptr,
    size_type offset,
    size_type count) const;

size_type find(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = 0) const;
```

Parameters

`char_value`

The character value for which the member function is to search.

`offset`

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

C++

```
// basic_string_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "Hello Everyone" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;

   indexCh1a = str1.find ( "e" , 3 );
   if (indexCh1a != string::npos )
      cout << "The index of the 1st 'e' found after the 3rd"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'e' was not found in str1 ." << endl;

   indexCh1b = str1.find ( "x" );
   if (indexCh1b != string::npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The Character 'x' was not found in str1."
           << endl << endl;
```

```
   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "Let me make this perfectly clear." );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "perfect";
   indexCh2a = str2.find ( cstr2 , 5 );
   if ( indexCh2a != string::npos )
      cout << "The index of the 1st element of 'perfect' "
           << "after\n the 5th position in str2 is: "
           << indexCh2a << endl;
   else
      cout << "The substring 'perfect' was not found in str2 ."
           << endl;

   const char *cstr2b = "imperfectly";
   indexCh2b = str2.find ( cstr2b , 0 );
   if (indexCh2b != string::npos )
      cout << "The index of the 1st element of 'imperfect' "
           << "after\n the 5th position in str3 is: "
           << indexCh2b << endl;
   else
      cout << "The substring 'imperfect' was not found in str2 ."
           << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "This is a sample string for this program" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "sample";
   indexCh3a = str3.find ( cstr3a );
   if ( indexCh3a != string::npos )
      cout << "The index of the 1st element of sample "
           << "in str3 is: " << indexCh3a << endl;
   else
      cout << "The substring 'sample' was not found in str3 ."
           << endl;

   const char *cstr3b = "for";
   indexCh3b = str3.find ( cstr3b , indexCh3a + 1 , 2 );
   if (indexCh3b != string::npos )
      cout << "The index of the next occurrence of 'for' is in "
           << "str3 begins at: " << indexCh3b << endl << endl;
   else
      cout << "There is no next occurrence of 'for' in str3 ."
           << endl << endl;

   // The fourth member function searches a string
   // for a substring as specified by a string
   string str4 ( "clearly this perfectly unclear." );
   cout << "The original string str4 is: " << str4 << endl;
   basic_string <char>::size_type indexCh4a, indexCh4b;
```

```
   string str4a ( "clear" );
   indexCh4a = str4.find ( str4a , 5 );
   if ( indexCh4a != string::npos )
      cout << "The index of the 1st element of 'clear' "
           << "after\n the 5th position in str4 is: "
           << indexCh4a << endl;
   else
      cout << "The substring 'clear' was not found in str4 ."
           << endl;

   string str4b ( "clear" );
   indexCh4b = str4.find ( str4b );
   if (indexCh4b != string::npos )
      cout << "The index of the 1st element of 'clear' "
           << "in str4 is: "
           << indexCh4b << endl;
   else
      cout << "The substring 'clear' was not found in str4 ."
           << endl << endl;
}
```

Output

```
The original string str1 is: Hello Everyone
The index of the 1st 'e' found after the 3rd position in str1 is: 8
The Character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
The index of the 1st element of 'perfect' after
the 5th position in str2 is: 17
The substring 'imperfect' was not found in str2 .

The original string str3 is: This is a sample string for this program
The index of the 1st element of sample in str3 is: 10
The index of the next occurrence of 'for' is in str3 begins at: 24

The original string str4 is: clearly this perfectly unclear.
The index of the 1st element of 'clear' after
the 5th position in str4 is: 25
The index of the 1st element of 'clear' in str4 is: 0
```

basic_string::find_first_not_of

Searches through a string for the first character that isn't an element of a specified string.

C++

```
size_type find_first_not_of(
    value_type char_value,
    size_type offset = 0) const;

size_type find_first_not_of(
    const value_type* ptr,
    size_type offset = 0) const;

size_type find_first_not_of(
    const value_type* ptr,
    size_type offset,
    size_type count) const;

size_type find_first_not_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = 0) const;
```

Parameters

char_value

The character value for which the member function is to search.

offset

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

C++

```cpp
// basic_string_find_first_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "xddd-1234-abcd" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_first_not_of ( "d" , 2 );
   if ( indexCh1a != npos )
      cout << "The index of the 1st 'd' found after the 3rd"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'd' was not found in str1 ." << endl;

   indexCh1b = str1.find_first_not_of  ( "x" );
   if (indexCh1b != npos )
      cout << "The index of the 'non x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'non x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "BBB-1111" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_first_not_of ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'B1' in str2 after\n the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements of the substring 'B1' were not"
           << "\n found in str2 after the 6th position."
           << endl;

   const char *cstr2b = "B2";
   indexCh2b = str2.find_first_not_of ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the 1st element of 'B2' "
           << "after\n the 0th position in str2 is: "
           << indexCh2b << endl << endl;
```

```
   else
      cout << "The substring 'B2' was not found in str2 ."
           << endl << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "444-555-GGG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "45G";
   indexCh3a = str3.find_first_not_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element in str3\n other than one of the "
           << "characters in '45G' is: " << indexCh3a
           << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string '45G'. "
           << endl;

   const char *cstr3b = "45G";
   indexCh3b = str3.find_first_not_of ( cstr3b , indexCh3a + 1 , 2 );
   if ( indexCh3b != npos )
      cout << "The index of the second occurrence of an "
           << "element of '45G' in str3\n after the 0th "
           << "position is: " << indexCh3b << endl << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string  '45G'. "
           << endl  << endl;

   // The fourth member function searches a string
   // for a substring as specified by a string
   string str4 ( "12-ab-12-ab" );
   cout << "The original string str4 is: " << str4 << endl;
   basic_string <char>::size_type indexCh4a, indexCh4b;

   string str4a ( "ba3" );
   indexCh4a = str4.find_first_not_of ( str4a , 5 );
   if (indexCh4a != npos )
      cout << "The index of the 1st non occurrence of an "
           << "element of 'ba3' in str4 after\n the 5th "
           << "position is: " << indexCh4a << endl;
   else
      cout << "Elements other than those in the substring"
           << " 'ba3' were not found in the string str4."
           << endl;

   string str4b ( "12" );
   indexCh4b = str4.find_first_not_of ( str4b  );
   if (indexCh4b != npos )
      cout << "The index of the 1st non occurrence of an "
           << "element of '12' in str4 after\n the 0th "
```

```
            << "position is: " << indexCh4b << endl;
   else
      cout << "Elements other than those in the substring"
           << " '12' were not found in the string str4."
           << endl;
}
```

Output

```
The original string str1 is: xddd-1234-abcd
The index of the 1st 'd' found after the 3rd position in str1 is: 4
The index of the 'non x' found in str1 is: 1

The original string str2 is: BBB-1111
Elements of the substring 'B1' were not
found in str2 after the 6th position.
The index of the 1st element of 'B2' after
the 0th position in str2 is: 3

The original string str3 is: 444-555-GGG
The index of the 1st occurrence of an element in str3
other than one of the characters in '45G' is: 3
The index of the second occurrence of an element of '45G' in str3
after the 0th position is: 7

The original string str4 is: 12-ab-12-ab
The index of the 1st non occurrence of an element of 'ba3' in str4 after
the 5th position is: 5
The index of the 1st non occurrence of an element of '12' in str4 after
the 0th position is: 2
```

basic_string::find_first_of

Searches through a string for the first character that matches any element of a specified string.

C++

```
size_type find_first_of(
    value_type char_value,
    size_type offset = 0) const;

size_type find_first_of(
    const value_type* ptr,
    size_type offset = 0) const;

size_type find_first_of(
    const value_type* ptr,
    size_type offset,
    size_type count) const;
```

```
size_type find_first_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = 0) const;
```

Parameters

char_value

The character value for which the member function is to search.

offset

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

C++

```
// basic_string_find_first_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "abcd-1234-abcd-1234" );
   cout << "The original string str1 is: " << str1 << endl;
```

```
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_first_of ( "d" , 5 );
   if ( indexCh1a != npos )
      cout << "The index of the 1st 'd' found after the 5th"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'd' was not found in str1 ." << endl;

   indexCh1b = str1.find_first_of ( "x" );
   if ( indexCh1b != npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for any element of a substring as specified by a C-string
   string str2 ( "ABCD-1234-ABCD-1234" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_first_of ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'B1' in str2 after\n the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements of the substring 'B1' were not "
           << "found in str2 after the 10th position."
           << endl;

   const char *cstr2b = "D2";
   indexCh2b = str2.find_first_of ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the 1st element of 'D2' "
           << "after\n the 0th position in str2 is: "
           << indexCh2b << endl << endl;
   else
      cout << "The substring 'D2' was not found in str2 ."
           << endl << endl << endl;

   // The third member function searches a string
   // for any element of a substring as specified by a C-string
   string str3 ( "123-abc-123-abc-456-EFG-456-EFG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "5G";
   indexCh3a = str3.find_first_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the 1st occurrence of an "
```

```
            << "element of '5G' in str3 after\n the 0th "
            << "position is: " << indexCh3a << endl;
   else
      cout << "Elements of the substring '5G' were not "
           << "found in str3\n after the 0th position."
           << endl;

   const char *cstr3b = "5GF";
   indexCh3b = str3.find_first_of  ( cstr3b , indexCh3a + 1 , 2 );
   if (indexCh3b != npos )
      cout << "The index of the second occurrence of an "
           << "element of '5G' in str3\n after the 0th "
           << "position is: " << indexCh3b << endl << endl;
   else
      cout << "Elements of the substring '5G' were not "
           << "found in str3\n after the first occurrrence."
           << endl << endl;

   // The fourth member function searches a string
   // for any element of a substring as specified by a string
   string str4 ( "12-ab-12-ab" );
   cout << "The original string str4 is: " << str4 << endl;
   basic_string <char>::size_type indexCh4a, indexCh4b;

   string str4a ( "ba3" );
   indexCh4a = str4.find_first_of ( str4a , 5 );
   if ( indexCh4a != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'ba3' in str4 after\n the 5th "
           << "position is: " << indexCh4a << endl;
   else
      cout << "Elements of the substring 'ba3' were not "
           << "found in str4\n after the 0th position."
           << endl;

   string str4b ( "a2" );
   indexCh4b = str4.find_first_of ( str4b );
   if ( indexCh4b != npos )
      cout << "The index of the 1st occurrence of an "
           << "element of 'a2' in str4 after\n the 0th "
           << "position is: " << indexCh4b << endl;
   else
      cout << "Elements of the substring 'a2' were not "
           << "found in str4\n after the 0th position."
           << endl;
}
```

Output

```
The original string str1 is: abcd-1234-abcd-1234
The index of the 1st 'd' found after the 5th position in str1 is: 13
The character 'x' was not found in str1.
```

```
The original string str2 is: ABCD-1234-ABCD-1234
The index of the 1st occurrence of an element of 'B1' in str2 after
the 6th position is: 11
The index of the 1st element of 'D2' after
the 0th position in str2 is: 3

The original string str3 is: 123-abc-123-abc-456-EFG-456-EFG
The index of the 1st occurrence of an element of '5G' in str3 after
the 0th position is: 17
The index of the second occurrence of an element of '5G' in str3
after the 0th position is: 22

The original string str4 is: 12-ab-12-ab
The index of the 1st occurrence of an element of 'ba3' in str4 after
the 5th position is: 9
The index of the 1st occurrence of an element of 'a2' in str4 after
the 0th position is: 1
```

basic_string::find_last_not_of

Searches through a string for the last character that isn't any element of a specified string.

C++

```
size_type find_last_not_of(
    value_type char_value,
    size_type offset = npos) const;

size_type find_last_not_of(
    const value_type* ptr,
    size_type offset = npos) const;

size_type find_last_not_of(
    const value_type* ptr,
    size_type offset,
    size_type count) const;

size_type find_last_not_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = npos) const;
```

Parameters

char_value

The character value for which the member function is to search.

offset

Index of the position at which the search is to finish.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

Example

C++

```
// basic_string_find_last_not_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "dddd-1dd4-abdd" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_last_not_of ( "d" , 7 );
   if ( indexCh1a != npos )
      cout << "The index of the last non 'd'\n found before the "
           << "7th position in str1 is: " << indexCh1a << endl;
   else
      cout << "The non 'd' character was not found ." << endl;

   indexCh1b = str1.find_last_not_of  ( "d" );
   if ( indexCh1b != npos )
      cout << "The index of the non 'd' found in str1 is: "
```

```
         << indexCh1b << endl << endl;
   else
      cout << "The Character 'non x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "BBB-1111" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
   indexCh2a = str2.find_last_not_of  ( cstr2 , 6 );
   if ( indexCh2a != npos )
      cout << "The index of the last occurrence of a "
           << "element\n not of 'B1' in str2 before the 6th "
           << "position is: " << indexCh2a << endl;
   else
      cout << "Elements not of the substring 'B1' were not "
           << "\n found in str2 before the 6th position."
           << endl;

   const char *cstr2b = "B-1";
   indexCh2b = str2.find_last_not_of  ( cstr2b );
   if ( indexCh2b != npos )
      cout << "The index of the last element not "
           << "in 'B-1'\n is: "
           << indexCh2b << endl << endl;
   else
      cout << "The elements of the substring 'B-1' were "
           << "not found in str2 ."
           << endl << endl;

   // The third member function searches a string
   // for a substring as specified by a C-string
   string str3 ( "444-555-GGG" );
   cout << "The original string str3 is: " << str3 << endl;
   basic_string <char>::size_type indexCh3a, indexCh3b;

   const char *cstr3a = "45G";
   indexCh3a = str3.find_last_not_of ( cstr3a );
   if ( indexCh3a != npos )
      cout << "The index of the last occurrence of an "
           << "element in str3\n other than one of the "
           << "characters in '45G' is: " << indexCh3a
           << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string  '45G'. "
           << endl;

   const char *cstr3b = "45G";
   indexCh3b = str3.find_last_not_of ( cstr3b , 6 , indexCh3a - 1 );
   if (indexCh3b != npos )
      cout << "The index of the penultimate occurrence of an "
```

```
            << "element\n not in '45G' in str3 is: "
            << indexCh3b << endl << endl;
   else
      cout << "Elements in str3 contain only characters "
           << " in the string '45G'. "
           << endl  << endl;

   // The fourth member function searches a string
   // for a substring as specified by a string
   string str4 ( "12-ab-12-ab" );
   cout << "The original string str4 is: " << str4 << endl;
   basic_string <char>::size_type indexCh4a, indexCh4b;

   string str4a ( "b-a" );
   indexCh4a = str4.find_last_not_of  ( str4a , 5 );
   if ( indexCh4a != npos )
      cout << "The index of the last occurrence of an "
           << "element not\n in 'b-a' in str4 before the 5th "
           << "position is: " << indexCh4a << endl;
   else
      cout << "Elements other than those in the substring"
           << " 'b-a' were not found in the string str4."
           << endl;

   string str4b ( "12" );
   indexCh4b = str4.find_last_not_of ( str4b  );
   if ( indexCh4b != npos )
      cout << "The index of the last occurrence of an "
           << "element not in '12'\n in str4 before the end "
           << "position is: " << indexCh4b << endl;
   else
      cout << "Elements other than those in the substring"
           << " '12'\n were not found in the string str4."
           << endl;
}
```

Output

```
The original string str1 is: dddd-1dd4-abdd
The index of the last non 'd'
found before the 7th position in str1 is: 5
The index of the non 'd' found in str1 is: 11

The original string str2 is: BBB-1111
The index of the last occurrence of a element
not of 'B1' in str2 before the 6th position is: 3
The elements of the substring 'B-1' were not found in str2 .

The original string str3 is: 444-555-GGG
The index of the last occurrence of an element in str3
other than one of the characters in '45G' is: 7
The index of the penultimate occurrence of an element
not in '45G' in str3 is: 3
```

```
The original string str4 is: 12-ab-12-ab
The index of the last occurrence of an element not
in 'b-a' in str4 before the 5th position is: 1
The index of the last occurrence of an element not in '12'
in str4 before the end position is: 10
```

`basic_string::find_last_of`

Searches through a string for the last character that matches any element of a specified string.

C++

```
size_type find_last_of(
    value_type char_value,
    size_type offset = npos) const;

size_type find_last_of(
    const value_type* ptr,
    size_type offset = npos) const;

size_type find_last_of(
    const value_type* ptr,
    size_type offset,
    size_type count) const;

size_type find_last_of(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = npos) const;
```

Parameters

`char_value`

The character value for which the member function is to search.

`offset`

Index of the position at which the search is to finish.

`ptr`

The C-string for which the member function is to search.

`count`

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the last character of the substring searched for when successful; otherwise `npos`.

Example

C++

```
// basic_string_find_last_of.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "abcd-1234-abcd-1234" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.find_last_of ( "d" , 14 );
   if ( indexCh1a != npos )
      cout << "The index of the last 'd' found before the 14th"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'd' was not found in str1 ." << endl;

   indexCh1b = str1.find_first_of ( "x" );
   if ( indexCh1b != npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'x' was not found in str1."
           << endl << endl;

   // The second member function searches a string
   // for a substring as specified by a C-string
   string str2 ( "ABCD-1234-ABCD-1234" );
   cout << "The original string str2 is: " << str2 << endl;
   basic_string <char>::size_type indexCh2a, indexCh2b;

   const char *cstr2 = "B1";
```

```
    indexCh2a = str2.find_last_of  ( cstr2 , 12 );
    if (indexCh2a != npos )
       cout << "The index of the last occurrence of an "
            << "element of 'B1' in str2 before\n the 12th "
            << "position is: " << indexCh2a << endl;
    else
       cout << "Elements of the substring 'B1' were not "
            << "found in str2 before the 12th position."
            << endl;

    const char *cstr2b = "D2";
    indexCh2b = str2.find_last_of  ( cstr2b );
    if ( indexCh2b != npos )
       cout << "The index of the last element of 'D2' "
            << "after\n the 0th position in str2 is: "
            << indexCh2b << endl << endl;
    else
       cout << "The substring 'D2' was not found in str2 ."
            << endl << endl << endl;

    // The third member function searches a string
    // for a substring as specified by a C-string
    string str3 ( "456-EFG-456-EFG" );
    cout << "The original string str3 is: " << str3 << endl;
    basic_string <char>::size_type indexCh3a;

    const char *cstr3a = "5E";
    indexCh3a = str3.find_last_of ( cstr3a , 8 , 8 );
    if ( indexCh3a != npos )
       cout << "The index of the last occurrence of an "
            << "element of '5E' in str3 before\n the 8th "
            << "position is: " << indexCh3a << endl << endl;
    else
       cout << "Elements of the substring '5G' were not "
            << "found in str3\n before the 8th position."
            << endl << endl;

    // The fourth member function searches a string
    // for a substring as specified by a string
    string str4 ( "12-ab-12-ab" );
    cout << "The original string str4 is: " << str4 << endl;
    basic_string <char>::size_type indexCh4a, indexCh4b;

    string str4a ( "ba3" );
    indexCh4a = str4.find_last_of  ( str4a , 8 );
    if ( indexCh4a != npos )
       cout << "The index of the last occurrence of an "
            << "element of 'ba3' in str4 before\n the 8th "
            << "position is: " << indexCh4a << endl;
    else
       cout << "Elements of the substring 'ba3' were not "
            << "found in str4\n after the 0th position."
            << endl;

    string str4b ( "a2" );
```

```
   indexCh4b = str4.find_last_of ( str4b  );
   if ( indexCh4b != npos )
      cout << "The index of the last occurrence of an "
           << "element of 'a2' in str4 before\n the 0th "
           << "position is: " << indexCh4b << endl;
   else
      cout << "Elements of the substring 'a2' were not "
           << "found in str4\n after the 0th position."
           << endl;
}
```

Output

```
The original string str1 is: abcd-1234-abcd-1234
The index of the last 'd' found before the 14th position in str1 is: 13
The character 'x' was not found in str1.

The original string str2 is: ABCD-1234-ABCD-1234
The index of the last occurrence of an element of 'B1' in str2 before
the 12th position is: 11
The index of the last element of 'D2' after
the 0th position in str2 is: 16

The original string str3 is: 456-EFG-456-EFG
The index of the last occurrence of an element of '5E' in str3 before
the 8th position is: 4

The original string str4 is: 12-ab-12-ab
The index of the last occurrence of an element of 'ba3' in str4 before
the 8th position is: 4
The index of the last occurrence of an element of 'a2' in str4 before
the 0th position is: 9
```

`basic_string::front`

Returns a reference to the first element in a string.

C++

```
const_reference front() const;

reference front();
```

Return value

A reference to the first element of the string, which must be non-empty.

Remarks

`basic_string::get_allocator`

Returns a copy of the allocator object used to construct the string.

```
C++
```

```
allocator_type get_allocator() const;
```

Return value

The allocator used by the string.

Remarks

The member function returns the stored allocator object.

Allocators for the string class specify how the class manages storage. The default allocators supplied with container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ feature.

Example

```
C++
```

```
// basic_string_get_allocator.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects
   // that use the default allocator.
   string s1;
   basic_string <char> s2;
   basic_string <char, char_traits< char >, allocator< char > > s3;

   // s4 will use the same allocator class as s1
   basic_string <char> s4( s1.get_allocator ( ) );

   basic_string <char>::allocator_type xchar = s1.get_allocator( );
```

```
    // You can now call functions on the allocator class xchar used by s1
}
```

basic_string::insert

Inserts an element, several elements, or a range of elements into the string at a specified position.

C++

```
basic_string<CharType, Traits, Allocator>& insert(
    size_type position,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& insert(
    size_type position,
    const value_type* ptr,
    size_type count);

basic_string<CharType, Traits, Allocator>& insert(
    size_type position,
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& insert(
    size_type position,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset,
    size_type count);

basic_string<CharType, Traits, Allocator>& insert(
    size_type position,
    size_type count,
    value_type char_value);

iterator insert(
    iterator iter);

iterator insert(
    iterator iter,
    value_type char_value)l
template <class InputIterator>
void insert(
    iterator iter,
    InputIterator first,
    InputIterator last);

void insert(
    iterator iter,
    size_type count,
    value_type char_value);
```

```
void insert(
    iterator iter,
    const_pointer first,
    const_pointer last);

void insert(
    iterator iter,
    const_iterator first,
    const_iterator last);
```

Parameters

position

The index of the position behind the point of insertion the new characters.

ptr

The C-string to be wholly or partly inserted into the string.

count

The number of characters to be inserted.

str

The string to be wholly or partly inserted into the target string.

offset

The index of the part of the source string supplying the characters to be appended.

char_value

The character value of the elements to be inserted.

iter

An iterator addressing the position behind which a character is to be inserted.

first

An input iterator, `const_pointer`, or `const_iterator` addressing the first element in the source range to be inserted.

last

An input iterator, `const_pointer`, or `const_iterator` addressing the position of the one beyond the last element in the source range to be inserted.

Return value

The functions that return a value return either:

- A reference to the `basic_string` that contains the original string plus the new characters.
- An iterator at the start of the inserted character(s).

Example

C++

```
// basic_string_insert.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function inserting a C-string
   // at a given position
   basic_string <char> str1a ( "way" );
   const char *cstr1a = "a";
   str1a.insert ( 0, cstr1a );
   cout << "The string with a C-string inserted at position 0 is: "
        << str1a << "." << endl;

   // The second member function inserting a C-string
   // at a given position for a specified number of elements
   basic_string <char> str2a ( "Good" );
   const char *cstr2a = "Bye Bye Baby";
   str2a.insert ( 4, cstr2a ,3 );
   cout << "The string with a C-string inserted at the end is: "
        << str2a << "." << endl;

   // The third member function inserting a string
   // at a given position
   basic_string <char> str3a ( "Bye" );
   string str3b ( "Good" );
   str3a.insert ( 0, str3b );
   cout << "The string with a string inserted at position 0 is: "
        << str3a << "." << endl;

   // The fourth member function inserting part of
   // a string at a given position
   basic_string <char> str4a ( "Good " );
   string str4b ( "Bye Bye Baby" );
   str4a.insert ( 5, str4b , 8 , 4 );
   cout << "The string with part of a string inserted at position 4 is: "
        << str4a << "." << endl;

   // The fifth member function inserts a number of characters
   // at a specified position in the string
   string str5 ( "The number is: ." );
```

```
    str5.insert ( 15 , 3 , '3' );
    cout << "The string with characters inserted is: "
         << str5 << endl;

    // The sixth member function inserts a character
    // at a specified position in the string
    string str6 ( "ABCDFG" );
    basic_string <char>::iterator str6_Iter = ( str6.begin ( ) + 4 );
    str6.insert ( str6_Iter , 'e' );
    cout << "The string with a character inserted is: "
         << str6 << endl;

    // The seventh member function inserts a range
    // at a specified position in the string
    string str7a ( "ABCDHIJ" );
    string str7b ( "abcdefgh" );
    basic_string <char>::iterator str7a_Iter = (str7a.begin ( ) + 4 );
    str7a.insert ( str7a_Iter , str7b.begin ( ) + 4 , str7b.end ( ) -1 );
    cout << "The string with a character inserted from a range is: "
         << str7a << endl;

    // The eighth member function inserts a number of
    // characters at a specified position in the string
    string str8 ( "ABCDHIJ" );
    basic_string <char>::iterator str8_Iter = ( str8.begin ( ) + 4 );
    str8.insert ( str8_Iter , 3 , 'e' );
    cout << "The string with a character inserted from a range is: "
         << str8 << endl;
}
```

Output

```
The string with a C-string inserted at position 0 is: away.
The string with a C-string inserted at the end is: GoodBye.
The string with a string inserted at position 0 is: GoodBye.
The string with part of a string inserted at position 4 is: Good Baby.
The string with characters inserted is: The number is: 333.
The string with a character inserted is: ABCDeFG
The string with a character inserted from a range is: ABCDefgHIJ
The string with a character inserted from a range is: ABCDeeeHIJ
```

`basic_string::iterator`

A type that provides a random-access iterator that can access and read a `const` element in the string.

C++

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of a character and is used to iterate through a string in a forward direction.

Example

See the example for begin for an example of how to declare and use `iterator`.

`basic_string::length`

Returns the current number of elements in a string.

C++

```
size_type length() const;
```

Remarks

The member function is the same as size.

Example

C++

```
// basic_string_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );
```

```
    // Compare size, length, capacity & max_size of a string
    cout << "The current size of original string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The current length of original string str1 is: "
         << lenStr1 << "." << endl;
    cout << "The capacity of original string str1 is: "
         << capStr1 << "." << endl;
    cout << "The max_size of original string str1 is: "
         << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
         << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
         << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
         << max_sizeStr1 << "." << endl;
}
```

`basic_string::max_size`

Returns the maximum number of characters a string could contain.

C++

```
size_type max_size() const;
```

Return value

The maximum number of characters a string could contain.

Remarks

An exception of type length_error Class is thrown when an operation produces a string with a length greater than the maximum size.

Example

C++

```
// basic_string_max_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of original string str1 is: "
        << max_sizeStr1 << "." << endl << endl;

   str1.erase ( 6, 5 );
   cout << "The modified string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   lenStr1 = str1.length ( );
   capStr1 = str1.capacity ( );
   max_sizeStr1 = str1.max_size ( );

   // Compare size, length, capacity & max_size of a string
   // after erasing part of the original string
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of modified string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl;
   cout << "The max_size of modified string str1 is: "
        << max_sizeStr1 << "." << endl;
}
```

`basic_string::npos`

An unsigned integral value initialized to -1 that indicates either "not found" or "all remaining characters" when a search function fails.

```
C++
```

```
static const size_type npos = -1;
```

Remarks

When the return value is to be checked for the `npos` value, it might not work unless the return value is of type size_type and not either `int` or `unsigned`.

Example

See the example for find for an example of how to declare and use `npos`.

`basic_string::operator+=`

Appends characters to a string.

```
C++
```

```
basic_string<CharType, Traits, Allocator>& operator+=(
    value_type char_value);

basic_string<CharType, Traits, Allocator>& operator+=(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& operator+=(
    const basic_string<CharType, Traits, Allocator>& right);
```

Parameters

`char_value`
The character to be appended.

`ptr`
The characters of the C-string to be appended.

right

The characters of the string to be appended.

Return value

A reference to the string object that is being appended with the characters passed by the member function.

Remarks

Characters may be appended to a string using the `operator+=` or the member functions append or push_back. The `operator+=` appends single-argument values while the multiple argument append member function allows a specific part of a string to be specified for adding.

Example

C++

```
// basic_string_op_app.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // appending a single character to a string
   string str1a ( "Hello" );
   cout << "The original string str1 is: " << str1a << endl;
   str1a +=  '!' ;
   cout << "The string str1 appended with an exclamation is: "
        << str1a << endl << endl;

   // The second member function
   // appending a C-string to a string
   string  str1b ( "Hello " );
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b << endl;
   str1b +=  cstr1b;
   cout << "Appending the C-string cstr1b to string str1 gives: "
        << str1b << "." << endl << endl;

   // The third member function
   // appending one string to another in two ways,
   // comparing append and operator [ ]
```

```
    string str1d ( "Hello " ), str2d ( "Wide " ), str3d ( "World" );
    cout << "The string str2d is: " << str2d << endl;
    str1d.append ( str2d );
    cout << "The appended string str1d is: "
         << str1d << "." << endl;
    str1d += str3d;
    cout << "The doubly appended strig str1 is: "
         << str1d << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello
The string str1 appended with an exclamation is: Hello!

The C-string cstr1b is: Out There
Appending the C-string cstr1b to string str1 gives: Hello Out There.

The string str2d is: Wide
The appended string str1d is: Hello Wide .
The doubly appended strig str1 is: Hello Wide World.
```

basic_string::operator=

Assigns new character values to the contents of a string.

C++

```
basic_string<CharType, Traits, Allocator>& operator=(
    value_type char_value);

basic_string<CharType, Traits, Allocator>& operator=(
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& operator=(
    const basic_string<CharType, Traits, Allocator>& right);

basic_string<CharType, Traits, Allocator>& operator=(
    const basic_string<CharType, Traits, Allocator>&& right);
```

Parameters

char_value

The character value to be assigned.

ptr

A pointer to the characters of the C-string to be assigned to the target string.

right

The source string whose characters are to be assigned to the target string.

Return value

A reference to the string object that is being assigned new characters by the member function.

Remarks

The strings may be assigned new character values. The new value may be either a string and C-string or a single character. The `operator=` may be used if the new value can be described by a single parameter, otherwise the member function assign, which has multiple parameters, may be used to specify which part of the string is to be assigned to a target string.

Example

C++

```
// basic_string_op_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function assigning a
   // character of a certain value to a string
   string str1a ( "Hello " );
   str1a = '0';
   cout << "The string str1 assigned with the zero character is: "
        << str1a << endl << endl;

   // The second member function assigning the
   // characters of a C-string to a string
   string  str1b;
   const char *cstr1b = "Out There";
   cout << "The C-string cstr1b is: " << cstr1b <<  "." << endl;
   str1b = cstr1b;
   cout << "Assigning the C-string cstr1a to string str1 gives: "
        << str1b << "." << endl << endl;

   // The third member function assigning the characters
   // from one string to another string in two equivalent
```

```
    // ways, comparing the assign and operator =
    string str1c ( "Hello" ), str2c ( "Wide" ), str3c ( "World" );
    cout << "The original string str1 is: " << str1c << "." << endl;
    cout << "The string str2c is: " << str2c << "." << endl;
    str1c.assign ( str2c );
    cout << "The string str1 newly assigned with string str2c is: "
         << str1c << "." << endl;
    cout << "The string str3c is: " << str3c << "." << endl;
    str1c = str3c;
    cout << "The string str1 reassigned with string str3c is: "
         << str1c << "." << endl << endl;
}
```

Output

```
The string str1 assigned with the zero character is: 0

The C-string cstr1b is: Out There.
Assigning the C-string cstr1a to string str1 gives: Out There.

The original string str1 is: Hello.
The string str2c is: Wide.
The string str1 newly assigned with string str2c is: Wide.
The string str3c is: World.
The string str1 reassigned with string str3c is: World.
```

`basic_string::operator[]`

Provides a reference to the character with a specified index in a string.

C++

```
const_reference operator[](size_type offset) const;
reference operator[](size_type offset);
```

Parameters

`offset`

The index of the position of the element to be referenced.

Return value

A reference to the character of the string at the position specified by the parameter index.

Remarks

The first element of the string has an index of zero, and the following elements are indexed consecutively by the positive integers. It means that a string of length *n* has an *n*th element indexed by the number *n* - 1.

`operator[]` is faster than the member function at for providing read and write access to the elements of a string.

`operator[]` doesn't check whether the index passed as a parameter is valid, but the member function `at` does and so should be used in the validity isn't certain. An invalid index (an index less that zero or greater than or equal to the size of the string) passed to the member function `at` throws an out_of_range Class exception. An invalid index passed to `operator[]` results in undefined behavior, but the index equal to the length of the string is a valid index for const strings and the operator returns the null character when passed this index.

The reference returned may be invalidated by string reallocations or modifications for the non-`const` strings.

When compiling with _ITERATOR_DEBUG_LEVEL set to 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the string. For more information, see Checked Iterators.

Example

C++

```
// basic_string_op_ref.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Hello world" ), str2 ( "Goodbye world" );
   const string cstr1 ( "Hello there" ), cstr2 ( "Goodbye now" );
   cout << "The original string str1 is: " << str1 << endl;
   cout << "The original string str2 is: " << str2 << endl;

   // Element access to the non-const strings
   basic_string <char>::reference refStr1 = str1 [6];
   basic_string <char>::reference refStr2 = str2.at ( 3 );

   cout << "The character with an index of 6 in string str1 is: "
        << refStr1 << "." << endl;
```

```
   cout << "The character with an index of 3 in string str2 is: "
        << refStr2 << "." << endl;

   // Element access to the const strings
   basic_string <char>::const_reference crefStr1 = cstr1 [ cstr1.length ( )
];
   basic_string <char>::const_reference crefStr2 = cstr2.at ( 8 );

   if ( crefStr1 == '\0' )
      cout << "The null character is returned as a valid reference."
           << endl;
   else
      cout << "The null character is not returned." << endl;
   cout << "The character with index of 8 in the const string cstr2 is: "
        << crefStr2 << "." << endl;
}
```

`basic_string::pointer`

A type that provides a pointer to a character element in a string or character array.

C++

```
typedef typename allocator_type::pointer pointer;
```

Remarks

The type is a synonym for `allocator_type::pointer`.

For type `string`, it's equivalent to `char *`.

Example

C++

```
// basic_string_pointer.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   basic_string<char>::pointer pstr1a = "In Here";
   char *cstr1b = "Out There";
   cout << "The string pstr1a is: " << pstr1a <<  "." << endl;
```

```
    cout << "The C-string cstr1b is: " << cstr1b << "." << endl;
}
```

Output

```
The string pstr1a is: In Here.
The C-string cstr1b is: Out There.
```

`basic_string::pop_back`

Erases the last element of the string.

C++

```
void pop_back();
```

Remarks

This member function effectively calls `erase(size() - 1)` to erase the last element of the sequence, which must be non-empty.

`basic_string::push_back`

Adds an element to the end of the string.

C++

```
void push_back(value_type char_value);
```

Parameters

`char_value`
The character to be added to the end of the string.

Remarks

The member function effectively calls `insert( end, char_value )`. For more information, see insert and end.

Example

C++

```
// basic_string_push_back.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "abc" );
   basic_string <char>::iterator str_Iter, str1_Iter;

   cout << "The original string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;

   // str1.push_back ( 'd' );
   str1_Iter = str1.end ( );
   str1_Iter--;
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_Iter << endl;

   cout << "The modified string str1 is: ";
   for ( str_Iter = str1.begin( ); str_Iter != str1.end( ); str_Iter++ )
      cout << *str_Iter;
   cout << endl;
}
```

Output

```
The original string str1 is: abc
The last character-letter of the modified str1 is now: c
The modified string str1 is: abc
```

basic_string::rbegin

Returns an iterator to the first element in a reversed string.

C++

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return value

Returns a random-access iterator to the first element in a reversed string, addressing what would be the last element in the corresponding unreversed string.

Remarks

`rbegin` is used with a reversed string just as begin is used with a string.

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the string object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the string object can be modified.

`rbegin` can be used to initialize an iteration through a string backwards.

Example

C++

```
// basic_string_rbegin.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ( "Able was I ere I saw Elba" ), str2;
   basic_string <char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
   basic_string <char>::const_reverse_iterator str1_rcIter;

   str1_rIter = str1.rbegin ( );
   // str1_rIter--;
   cout << "The first character-letter of the reversed string str1 is: "
        << *str1_rIter << endl;
   cout << "The full reversed string str1 is:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++
)
      cout << *str_rIter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
   *str1_rIter = 'A';
   cout << "The first character-letter of the modified str1 is now: "
        << *str1_rIter << endl;
   cout << "The full modified reversed string str1 is now:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++
)
      cout << *str_rIter;
```

```
   cout << endl;

   // The following line would be an error because iterator is const
   // *str1_rcIter = 'A';

   // For an empty string, begin is equivalent to end
   if ( str2.rbegin( ) == str2.rend ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The first character-letter of the reversed string str1 is: a
The full reversed string str1 is:
ablE was I ere I saw elbA
The first character-letter of the modified str1 is now: A
The full modified reversed string str1 is now:
AblE was I ere I saw elbA
The string str2 is empty.
```

`basic_string::reference`

A type that provides a reference to an element stored in a string.

C++

```
typedef typename allocator_type::reference reference;
```

Remarks

A type `reference` can be used to modify the value of an element.

The type is a synonym for `allocator_type::reference`.

For type `string`, it's equivalent to `chr&`.

Example

See the example for at for an example of how to declare and use `reference`.

`basic_string::rend`

Returns an iterator that addresses the location succeeding the last element in a reversed string.

```
C++
```

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed string.

Remarks

`rend` is used with a reversed string just as end is used with a string.

If the return value of `rend` is assigned to a `const_reverse_iterator`, the string object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, the string object can be modified.

`rend` can be used to test whether a reverse iterator has reached the end of its string.

The value returned by `rend` shouldn't be dereferenced.

Example

```
C++
```

```
// basic_string_rend.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Able was I ere I saw Elba"), str2;
   basic_string <char>::reverse_iterator str_rIter, str1_rIter, str2_rIter;
   basic_string <char>::const_reverse_iterator str1_rcIter;

   str1_rIter = str1.rend ( );
   str1_rIter--;
   cout << "The last character-letter of the reversed string str1 is: "
        << *str1_rIter << endl;
```

```
   cout << "The full reversed string str1 is:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++
)
      cout << *str_rIter;
   cout << endl;

   // The dereferenced iterator can be used to modify a character
   *str1_rIter = 'o';
   cout << "The last character-letter of the modified str1 is now: "
        << *str1_rIter << endl;
   cout << "The full modified reversed string str1 is now:\n ";
   for ( str_rIter = str1.rbegin( ); str_rIter != str1.rend( ); str_rIter++
)
      cout << *str_rIter;
   cout << endl;

   // The following line would be an error because iterator is const
   // *str1_rcIter = 'T';

   // For an empty string, end is equivalent to begin
   if ( str2.rbegin( ) == str2.rend ( ) )
      cout << "The string str2 is empty." << endl;
   else
      cout << "The stringstr2  is not empty." << endl;
}
```

Output

```
The last character-letter of the reversed string str1 is: A
The full reversed string str1 is:
ablE was I ere I saw elbA
The last character-letter of the modified str1 is now: o
The full modified reversed string str1 is now:
ablE was I ere I saw elbo
The string str2 is empty.
```

basic_string::replace

Replaces elements in a string at a specified position with specified characters, or with characters copied from other ranges, strings, or C-strings.

C++

```
basic_string<CharType, Traits, Allocator>& replace(
    size_type position_1,
    size_type number_1,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& replace(
    size_type position_1,
```

```
    size_type number_1,
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& replace(
    size_type position_1,
    size_type number_1,
    const value_type* ptr,
    size_type number_2);

basic_string<CharType, Traits, Allocator>& replace(
    size_type position_1,
    size_type number_1,
    const basic_string<CharType, Traits, Allocator>& str,
    size_type position_2,
    size_type number_2);

basic_string<CharType, Traits, Allocator>& replace(
    size_type position_1,
    size_type number_1,
    size_type count,
    value_type char_value);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const value_type* ptr);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const basic_string<CharType, Traits, Allocator>& str);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const value_type* ptr,
    size_type number_2);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    size_type number_2,
    value_type char_value);

template <class InputIterator>
basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    InputIterator first,
    InputIterator last);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const_pointer first,
```

```
    const_pointer last);

basic_string<CharType, Traits, Allocator>& replace(
    iterator first0,
    iterator last0,
    const_iterator first,
    const_iterator last);
```

Parameters

str

The string that is to be a source of characters for the operand string.

position_1

The index of the operand string at which the replacement begins.

number_1

The maximum number of characters to be replaced in the operand string.

position_2

The index of the parameter string at which the copying begins.

number_2

The maximum number of characters to be used from the parameter C-string.

ptr

The C-string that is to be a source of characters for the operand string.

char_value

The character to be copied into the operand string.

first0

An iterator addressing the first character to be removed in the operand string.

last0

An iterator addressing the last character to be removed in the operand string.

first

An iterator, const_pointer, or const_iterator addressing the first character to be copied in the parameter string.

last

An iterator, const_pointer, or const_iterator addressing the last character to be copied in the parameter string.

count

The number of times *char_value* is copied into the operand string.

Return value

The operand string with the replacement made.

Example

C++

```
// basic_string_replace.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first two member functions replace
   // part of the operand string with
   // characters from a parameter string or C-string
   string result1a, result1b;
   string s1o ( "AAAAAAAA" );
   string s1p ( "BBB" );
   const char* cs1p = "CCC";
   cout << "The operand string s1o is: " << s1o << endl;
   cout << "The parameter string s1p is: " << s1p << endl;
   cout << "The parameter C-string cs1p is: " << cs1p << endl;
   result1a = s1o.replace ( 1 , 3 , s1p );
   cout << "The result of s1o.replace ( 1 , 3 , s1p )\n is "
        << "the string: " << result1a << "." << endl;
   result1b = s1o.replace ( 5 , 3 , cs1p );
   cout << "The result of s1o.replace ( 5 , 3 , cs1p )\n is "
        << "the string: " << result1b << "." << endl;
   cout << endl;

   // The third & fourth member function replace
   // part of the operand string with characters
   // form part of a parameter string or C-string
   string result2a, result2b;
   string s2o ( "AAAAAAAA" );
   string s2p ( "BBB" );
   const char* cs2p = "CCC";
   cout << "The operand string s2o is: " << s2o << endl;
   cout << "The parameter string s1p is: " << s2p << endl;
   cout << "The parameter C-string cs2p is: " << cs2p << endl;
   result2a = s2o.replace ( 1 , 3 , s2p , 1 , 2 );
   cout << "The result of s2o.replace (1, 3, s2p, 1, 2)\n is "
        << "the string: " << result2a << "." << endl;
```

```
    result2b = s2o.replace ( 4 , 3 , cs2p , 1 );
    cout << "The result of s2o.replace (4 ,3 ,cs2p)\n is "
         << "the string: " << result2b << "." << endl;
    cout << endl;

    // The fifth member function replaces
    // part of the operand string with characters
    string result3a;
    string s3o ( "AAAAAAAA" );
    char ch3p = 'C';
    cout << "The operand string s3o is: " << s3o << endl;
    cout << "The parameter character c1p is: " << ch3p << endl;
    result3a = s3o.replace ( 1 , 3 , 4 , ch3p );
    cout << "The result of s3o.replace(1, 3, 4, ch3p)\n is "
         << "the string: " << result3a << "." << endl;
    cout << endl;

    // The sixth & seventh member functions replace
    // part of the operand string, delineated with iterators,
    // with a parameter string or C-string
    string s4o ( "AAAAAAAA" );
    string s4p ( "BBB" );
    const char* cs4p = "CCC";
    cout << "The operand string s4o is: " << s4o << endl;
    cout << "The parameter string s4p is: " << s4p << endl;
    cout << "The parameter C-string cs4p is: " << cs4p << endl;
    basic_string<char>::iterator IterF0, IterL0;
    IterF0 = s4o.begin ( );
    IterL0 = s4o.begin ( ) + 3;
    string result4a, result4b;
    result4a = s4o.replace ( IterF0 , IterL0 , s4p );
    cout << "The result of s1o.replace (IterF0, IterL0, s4p)\n is "
         << "the string: " << result4a << "." << endl;
    result4b = s4o.replace ( IterF0 , IterL0 , cs4p );
    cout << "The result of s4o.replace (IterF0, IterL0, cs4p)\n is "
         << "the string: " << result4b << "." << endl;
    cout << endl;

    // The 8th member function replaces
    // part of the operand string delineated with iterators
    // with a number of characters from a parameter C-string
    string s5o ( "AAAAAAAF" );
    const char* cs5p = "CCCBB";
    cout << "The operand string s5o is: " << s5o << endl;
    cout << "The parameter C-string cs5p is: " << cs5p << endl;
    basic_string<char>::iterator IterF1, IterL1;
    IterF1 = s5o.begin ( );
    IterL1 = s5o.begin ( ) + 4;
    string result5a;
    result5a = s5o.replace ( IterF1 , IterL1 , cs5p , 4 );
    cout << "The result of s5o.replace (IterF1, IterL1, cs4p ,4)\n is "
         << "the string: " << result5a << "." << endl;
    cout << endl;

    // The 9th member function replaces
```

```
   // part of the operand string delineated with iterators
   // with specified characters
   string s6o ( "AAAAAAAG" );
   char ch6p = 'q';
   cout << "The operand string s6o is: " << s6o << endl;
   cout << "The parameter character ch6p is: " << ch6p << endl;
   basic_string<char>::iterator IterF2, IterL2;
   IterF2 = s6o.begin ( );
   IterL2 = s6o.begin ( ) + 3;
   string result6a;
   result6a = s6o.replace ( IterF2 , IterL2 , 4 , ch6p );
   cout << "The result of s6o.replace (IterF1, IterL1, 4, ch6p)\n is "
        << "the string: " << result6a << "." << endl;
   cout << endl;

   // The 10th member function replaces
   // part of the operand string delineated with iterators
   // with part of a parameter string delineated with iterators
   string s7o ( "OOOOOOO" );
   string s7p ( "PPPP" );
   cout << "The operand string s7o is: " << s7o << endl;
   cout << "The parameter string s7p is: " << s7p << endl;
   basic_string<char>::iterator IterF3, IterL3, IterF4, IterL4;
   IterF3 = s7o.begin ( ) + 1;
   IterL3 = s7o.begin ( ) + 3;
   IterF4 = s7p.begin ( );
   IterL4 = s7p.begin ( ) + 2;
   string result7a;
   result7a = s7o.replace ( IterF3 , IterL3 , IterF4 , IterL4 );
   cout << "The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)\n is
"
        << "the string: " << result7a << "." << endl;
   cout << endl;
}
```

Output

```
The operand string s1o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs1p is: CCC
The result of s1o.replace ( 1 , 3 , s1p )
is the string: ABBBAAAA.
The result of s1o.replace ( 5 , 3 , cs1p )
is the string: ABBBACCC.

The operand string s2o is: AAAAAAAA
The parameter string s1p is: BBB
The parameter C-string cs2p is: CCC
The result of s2o.replace (1, 3, s2p, 1, 2)
is the string: ABBAAAA.
The result of s2o.replace (4 ,3 ,cs2p)
is the string: ABBAC.
```

```
The operand string s3o is: AAAAAAAA
The parameter character c1p is: C
The result of s3o.replace(1, 3, 4, ch3p)
is the string: ACCCCAAAA.

The operand string s4o is: AAAAAAAA
The parameter string s4p is: BBB
The parameter C-string cs4p is: CCC
The result of s1o.replace (IterF0, IterL0, s4p)
is the string: BBBAAAAA.
The result of s4o.replace (IterF0, IterL0, cs4p)
is the string: CCCAAAAA.

The operand string s5o is: AAAAAAAF
The parameter C-string cs5p is: CCCBB
The result of s5o.replace (IterF1, IterL1, cs4p ,4)
is the string: CCCBAAAF.

The operand string s6o is: AAAAAAAG
The parameter character ch6p is: q
The result of s6o.replace (IterF1, IterL1, 4, ch6p)
is the string: qqqqAAAAG.

The operand string s7o is: OOOOOOO
The parameter string s7p is: PPPP
The result of s7o.replace (IterF3 ,IterL3 ,IterF4 ,IterL4)
is the string: OPPOOOO.
```

`basic_string::reserve`

Sets the capacity of the string to a number at least as great as a specified number.

C++

```
void reserve(size_type count = 0);
```

Parameters

`count`

The number of characters for which memory is being reserved.

Remarks

Having sufficient capacity is important because reallocation is a time-consuming process. And, it invalidates all references, pointers, and iterators that refer to characters in a string.

The concept of capacity for string object types is the same as for objects of type `vector`. Unlike `vector`, the member function `reserve` may be called to shrink the capacity of an object. The request is nonbinding and may or may not happen. As the default value for the parameter is zero, a call of `reserve` is a non-binding request to shrink the capacity of the string to fit the number of characters currently in the string. The capacity is never reduced below the current number of characters.

Calling `reserve` is the only possible way to shrink the capacity of a string. However, as noted above, this request is nonbinding and may not happen.

Example

C++

```
// basic_string_reserve.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   basic_string <char>::size_type sizeStr1, sizerStr1;
   sizeStr1 = str1.size ( );
   basic_string <char>::size_type capStr1, caprStr1;
   capStr1 = str1.capacity ( );

   // Compare size & capacity of the original string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

   // Compare size & capacity of the string
   // with added capacity
   str1.reserve ( 40 );
   sizerStr1 = str1.size ( );
   caprStr1 = str1.capacity ( );

   cout << "The string str1with augmented capacity is: "
        << str1 << endl;
   cout << "The current size of string str1 is: "
        << sizerStr1 << "." << endl;
   cout << "The new capacity of string str1 is: "
        << caprStr1 << "." << endl << endl;

   // Compare size & capacity of the string
```

```
   // with downsized capacity
   str1.reserve ( );
   basic_string <char>::size_type sizedStr1;
   basic_string <char>::size_type capdStr1;
   sizedStr1 = str1.size ( );
   capdStr1 = str1.capacity ( );

   cout << "The string str1 with downsized capacity is: "
        << str1 << endl;
   cout << "The current size of string str1 is: "
        << sizedStr1 << "." << endl;
   cout << "The reduced capacity of string str1 is: "
        << capdStr1 << "." << endl << endl;
}
```

Output

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.

The string str1with augmented capacity is: Hello world
The current size of string str1 is: 11.
The new capacity of string str1 is: 47.

The string str1 with downsized capacity is: Hello world
The current size of string str1 is: 11.
The reduced capacity of string str1 is: 47.
```

basic_string::resize

Specifies a new size for a string, appending or erasing elements as required.

C++

```
void resize(
    size_type count,);

void resize(
    size_type count,
    value_type char_value);
```

Parameters

count

The new size of the string.

char_value

The value that appended characters are initialized with, if more elements are required.

Remarks

If the resulting size exceeds the maximum number of characters, the form throws `length_error`.

Example

C++

```
// basic_string_resize.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string  str1 ( "Hello world" );
   cout << "The original string str1 is: " << str1 << endl;

   basic_string <char>::size_type sizeStr1;
   sizeStr1 = str1.size ( );
   basic_string <char>::size_type capStr1;
   capStr1 = str1.capacity ( );

   // Compare size & capacity of the original string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl << endl;

   // Use resize to increase size by 2 elements: exclamations
   str1.resize ( str1.size ( ) + 2 , '!' );
   cout << "The resized string str1 is: " << str1 << endl;

   sizeStr1 = str1.size ( );
   capStr1 = str1.capacity ( );

   // Compare size & capacity of a string after resizing
   cout << "The current size of resized string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of resized string str1 is: "
        << capStr1 << "." << endl << endl;

   // Use resize to increase size by 20 elements:
   str1.resize ( str1.size ( ) + 20 );
   cout << "The resized string str1 is: " << str1 << endl;
```

```
   sizeStr1 = str1.size ( );
   capStr1 = str1.capacity ( );

   // Compare size & capacity of a string after resizing
   // note capacity increases automatically as required
   cout << "The current size of modified string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of modified string str1 is: "
        << capStr1 << "." << endl << endl;

   // Use resize to downsize by 28 elements:
   str1.resize ( str1.size ( ) - 28 );
   cout << "The downsized string str1 is: " << str1 << endl;

   sizeStr1 = str1.size (  );
   capStr1 = str1.capacity (  );

   // Compare size & capacity of a string after downsizing
   cout << "The current size of downsized string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of downsized string str1 is: "
        << capStr1 << "." << endl;
}
```

Output

```
The original string str1 is: Hello world
The current size of original string str1 is: 11.
The capacity of original string str1 is: 15.

The resized string str1 is: Hello world!!
The current size of resized string str1 is: 13.
The capacity of resized string str1 is: 15.

The resized string str1 is: Hello world!!
The current size of modified string str1 is: 33.
The capacity of modified string str1 is: 47.

The downsized string str1 is: Hello
The current size of downsized string str1 is: 5.
The capacity of downsized string str1 is: 47.
```

basic_string::reverse_iterator

A type that provides a reference to an element stored in a string.

C++

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` can be used to modify the value of a character and is used to iterate through a string in reverse.

Example

See the example for rbegin for an example of how to declare and use `reverse_iterator`.

`basic_string::rfind`

Searches a string in a backward direction for the first occurrence of a substring that matches a specified sequence of characters.

C++

```
size_type rfind(
    value_type char_value,
    size_type offset = npos) const;

size_type rfind(
    const value_type* ptr,
    size_type offset = npos) const;

size_type rfind(
    const value_type* ptr,
    size_type offset,
    size_type count) const;

size_type rfind(
    const basic_string<CharType, Traits, Allocator>& str,
    size_type offset = npos) const;
```

Parameters

`char_value`

The character value for which the member function is to search.

`offset`

Index of the position at which the search is to begin.

ptr

The C-string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C-string for which the member function is to search.

str

The string for which the member function is to search.

Return value

The index of the last occurrence, when searched backwards, of the first character of the substring when successful; otherwise `npos`.

Example

C++

```
// basic_string_rfind.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // The first member function
   // searches for a single character in a string
   string str1 ( "Hello Everyone" );
   cout << "The original string str1 is: " << str1 << endl;
   basic_string <char>::size_type indexCh1a, indexCh1b;
   static const basic_string <char>::size_type npos = -1;

   indexCh1a = str1.rfind ( "e" , 9 );
   if ( indexCh1a != npos )
      cout << "The index of the 1st 'e' found before the 9th"
           << " position in str1 is: " << indexCh1a << endl;
   else
      cout << "The character 'e' was not found in str1 ." << endl;

   indexCh1b = str1.rfind ( "x" );
   if ( indexCh1b != npos )
      cout << "The index of the 'x' found in str1 is: "
           << indexCh1b << endl << endl;
   else
      cout << "The character 'x' was not found in str1."
           << endl << endl;
```

```
// The second member function searches a string
// for a substring as specified by a C-string
string str2 ( "Let me make this perfectly clear." );
cout << "The original string str2 is: " << str2 << endl;
basic_string <char>::size_type indexCh2a, indexCh2b;

const char *cstr2 = "perfect";
indexCh2a = str2.rfind ( cstr2 , 30 );
if ( indexCh2a != npos )
   cout << "The index of the 1st element of 'perfect' "
        << "before\n the 30th position in str2 is: "
        << indexCh2a << endl;
else
   cout << "The substring 'perfect' was not found in str2 ."
        << endl;

const char *cstr2b = "imperfectly";
indexCh2b = str2.rfind ( cstr2b , 30 );
if ( indexCh2b != npos )
   cout << "The index of the 1st element of 'imperfect' "
        << "before\n the 5th position in str3 is: "
        << indexCh2b << endl;
else
   cout << "The substring 'imperfect' was not found in str2 ."
        << endl << endl;

// The third member function searches a string
// for a substring as specified by a C-string
string str3 ( "It is a nice day. I am happy." );
cout << "The original string str3 is: " << str3 << endl;
basic_string <char>::size_type indexCh3a, indexCh3b;

const char *cstr3a = "nice";
indexCh3a = str3.rfind ( cstr3a );
if ( indexCh3a != npos )
   cout << "The index of the 1st element of 'nice' "
        << "in str3 is: " << indexCh3a << endl;
else
   cout << "The substring 'nice' was not found in str3 ."
        << endl;

const char *cstr3b = "am";
indexCh3b = str3.rfind ( cstr3b , indexCh3a + 25 , 2 );
if ( indexCh3b != npos )
   cout << "The index of the next occurrence of 'am' in "
        << "str3 begins at: " << indexCh3b << endl << endl;
else
   cout << "There is no next occurrence of 'am' in str3 ."
        << endl << endl;

// The fourth member function searches a string
// for a substring as specified by a string
string str4 ( "This perfectly unclear." );
cout << "The original string str4 is: " << str4 << endl;
```

```
   basic_string <char>::size_type indexCh4a, indexCh4b;

   string str4a ( "clear" );
   indexCh4a = str4.rfind ( str4a , 15 );
   if (indexCh4a != npos )
      cout << "The index of the 1st element of 'clear' "
           << "before\n the 15th position in str4 is: "
           << indexCh4a << endl;
   else
      cout << "The substring 'clear' was not found in str4 "
           << "before the 15th position." << endl;

   string str4b ( "clear" );
   indexCh4b = str4.rfind ( str4b );
   if ( indexCh4b != npos )
      cout << "The index of the 1st element of 'clear' "
           << "in str4 is: "
           << indexCh4b << endl;
   else
      cout << "The substring 'clear' was not found in str4 ."
           << endl << endl;
}
```

Output

```
The original string str1 is: Hello Everyone
The index of the 1st 'e' found before the 9th position in str1 is: 8
The character 'x' was not found in str1.

The original string str2 is: Let me make this perfectly clear.
The index of the 1st element of 'perfect' before
the 30th position in str2 is: 17
The substring 'imperfect' was not found in str2 .

The original string str3 is: It is a nice day. I am happy.
The index of the 1st element of 'nice' in str3 is: 8
The index of the next occurrence of 'am' in str3 begins at: 20

The original string str4 is: This perfectly unclear.
The substring 'clear' was not found in str4 before the 15th position.
The index of the 1st element of 'clear' in str4 is: 17
```

basic_string::shrink_to_fit

Discards the excess capacity of the string.

C++

```
void shrink_to_fit();
```

Remarks

This member function eliminates any unneeded storage in the container.

`basic_string::size`

Returns the current number of elements in a string.

```
C++
```

```
size_type size() const;
```

Return value

The length of the string.

Example

```
C++
```

```
// basic_string_size.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string str1 ("Hello world");
   cout << "The original string str1 is: " << str1 << endl;

   // The size and length member functions differ in name only
   basic_string <char>::size_type sizeStr1, lenStr1;
   sizeStr1 = str1.size (  );
   lenStr1 = str1.length (  );

   basic_string <char>::size_type capStr1, max_sizeStr1;
   capStr1 = str1.capacity (  );
   max_sizeStr1 = str1.max_size (  );

   // Compare size, length, capacity & max_size of a string
   cout << "The current size of original string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The current length of original string str1 is: "
        << lenStr1 << "." << endl;
   cout << "The capacity of original string str1 is: "
        << capStr1 << "." << endl;
```

```
    cout << "The max_size of original string str1 is: "
         << max_sizeStr1 << "." << endl << endl;

    str1.erase ( 6, 5 );
    cout << "The modified string str1 is: " << str1 << endl;

    sizeStr1 = str1.size ( );
    lenStr1 = str1.length ( );
    capStr1 = str1.capacity ( );
    max_sizeStr1 = str1.max_size ( );

    // Compare size, length, capacity & max_size of a string
    // after erasing part of the original string
    cout << "The current size of modified string str1 is: "
         << sizeStr1 << "." << endl;
    cout << "The current length of modified string str1 is: "
         << lenStr1 << "." << endl;
    cout << "The capacity of modified string str1 is: "
         << capStr1 << "." << endl;
    cout << "The max_size of modified string str1 is: "
         << max_sizeStr1 << "." << endl;
}
```

`basic_string::size_type`

An unsigned integer type that can represent the number of elements and indices in a string.

C++

```
typedef typename allocator_type::size_type size_type;
```

Remarks

it's equivalent to `allocator_type::size_type`.

For type `string`, it's equivalent to `size_t`.

Example

C++

```
// basic_string_size_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   string str1 ( "Hello world" );

   basic_string <char>::size_type sizeStr1, capStr1;
   sizeStr1 = str1.size (  );
   capStr1 = str1.capacity (  );

   cout << "The current size of string str1 is: "
        << sizeStr1 << "." << endl;
   cout << "The capacity of string str1 is: " << capStr1
         << "." << endl;
}
```

Output

```
The current size of string str1 is: 11.
The capacity of string str1 is: 15.
```

`basic_string::starts_with`

Check whether the string starts with the specified prefix.

C++

```
bool starts_with(const CharType c) const noexcept;
bool starts_with(const CharType* const x) const noexcept;
bool starts_with(const basic_string_view sv) const noexcept;
```

Parameters

`c`

The single character prefix to look for.

`sv`

A string view containing the prefix to look for.
You can pass a `std::basic_string`, which converts to a string view.

`x`

Null-terminated character string containing the prefix to look for.

Return value

`true` if the string starts with the specified prefix; `false` otherwise.

Remarks

`starts_with()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

See ends_with to see if a string ends with the specified suffix.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <string>
#include <iostream>

int main()
{
    std::basic_string<char> str = "abcdefg";

    std::cout << std::boolalpha; // so booleans show as 'true'/'false'
    std::cout << str.starts_with('b') << '\n';
    std::cout << str.starts_with("aBc") << '\n';

    std::basic_string<char> str2 = "abc";
    std::cout << str.starts_with(str2);

    return 0;
}
```

Output

```
false
false
true
```

`basic_string::substr`

Copies a substring of at most some number of characters from a string beginning from a specified position.

C++

```
basic_string<CharType, Traits, Allocator> substr(
    size_type offset = 0,
    size_type count = npos) const;
```

Parameters

offset

An index locating the element at the position from which the copy of the string is made, with a default value of 0.

count

The number of characters that are to be copied if they're present.

Return value

A substring object that's a copy of elements of the string operand, beginning at the position specified by the first argument.

Example

C++

```
// basic_string_substr.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   string  str1 ("Heterological paradoxes are persistent.");
   cout << "The original string str1 is: \n " << str1
        << endl << endl;

   basic_string <char> str2 = str1.substr ( 6 , 7 );
   cout << "The substring str1 copied is: " << str2
        << endl << endl;

   basic_string <char> str3 = str1.substr (  );
   cout << "The default substring str3 is: \n " << str3
        <<  "\n which is the entire original string." << endl;
}
```

Output

```
The original string str1 is:
Heterological paradoxes are persistent.

The substring str1 copied is: logical

The default substring str3 is:
Heterological paradoxes are persistent.
which is the entire original string.
```

`basic_string::swap`

Exchange the contents of two strings.

C++

```
void swap(
    basic_string<CharType, Traits, Allocator>& str);
```

Parameters

`str`
The source string whose elements are to be exchanged with the ones in the destination string.

Remarks

If the strings being swapped have the same allocator object, the `swap` member function:

- Occurs in constant time.
- Throws no exceptions.
- Invalidates no references, pointers, or iterators that designate elements in the two strings.

Otherwise, it makes element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

C++

```
// basic_string_swap.cpp
// compile with: /EHsc
```

```cpp
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   // Declaring an objects of type basic_string<char>
   string s1 ( "Tweedledee" );
   string s2 ( "Tweedledum" );
   cout << "Before swapping string s1 and s2:" << endl;
   cout << " The basic_string s1 = " << s1 << "." << endl;
   cout << " The basic_string s2 = " << s2 << "." << endl;

   s1.swap ( s2 );
   cout << "After swapping string s1 and s2:" << endl;
   cout << " The basic_string s1 = " << s1 << "." << endl;
   cout << " The basic_string s2 = " << s2 << "." << endl;
}
```

Output

```
Before swapping string s1 and s2:
The basic_string s1 = Tweedledee.
The basic_string s2 = Tweedledum.
After swapping string s1 and s2:
The basic_string s1 = Tweedledum.
The basic_string s2 = Tweedledee.
```

`basic_string::traits_type`

A type for the character traits of the elements stored in a string.

C++

```cpp
typedef Traits traits_type;
```

Remarks

The type is a synonym for the second template parameter `Traits`.

For type `string`, it's equivalent to `char_traits<char>`.

Example

See the example for copy for an example of how to declare and use `traits_type`.

`basic_string::value_type`

A type that represents the type of characters stored in a string.

C++

```
typedef typename allocator_type::value_type value_type;
```

Remarks

It's equivalent to `traits_type::char_type` and is equivalent to `char` for objects of type `string`.

Example

C++

```
// basic_string_value_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   basic_string<char>::value_type ch1 = 'G';

   char ch2 = 'H';

   cout << "The character ch1 is: " << ch1 << "." << endl;
   cout << "The character ch2 is: " << ch2 << "." << endl;
}
```

Output

```
The character ch1 is: G.
The character ch2 is: H.
```

See also

<string>
Thread safety in the C++ standard library

char_traits Struct

Article • 08/03/2021

The char_traits struct describes attributes associated with a character.

Syntax

```
C++
```

```
template <class CharType>
struct char_traits;
```

Parameters

CharType
The element data type.

Remarks

The template struct describes various character traits for type `CharType`. The class template basic_string as well as several iostream class templates, including basic_ios, use this information to manipulate elements of type `CharType`. Such an element type must not require explicit construction or destruction. It must supply a default constructor, a copy constructor, and an assignment operator, with the expected semantics. A bitwise copy must have the same effect as an assignment. None of the member functions of struct char_traits can throw exceptions.

Typedefs

Type name	Description
char_type	A type of character.
int_type	An integer type that can represent a character of type `char_type` or an end-of-file (EOF) character.
off_type	An integer type that can represent offsets between positions in a stream.
pos_type	An integer type that can represent positions in a stream.

Type name	Description
state_type	A type that represents the conversion state in for multibyte characters in a stream.

Member functions

Member function	Description
assign	Assigns one character value to another.
compare	Compares up to a specified number of characters in two strings.
copy	Copies a specified number of characters from one string to another. Deprecated. Use char_traits::_Copy_s instead.
_Copy_s	Copies a specified number of characters from one string to another.
eof	Returns the end-of-file (EOF) character.
eq	Tests whether two `char_type` characters are equal.
eq_int_type	Tests whether two characters represented as `int_type`s are equal.
find	Searches for the first occurrence of a specified character in a range of characters.
length	Returns the length of a string.
lt	Tests whether one character is less than another.
move	Copies a specified number of characters in a sequence to another, possible overlapping, sequence. Deprecated. Use char_traits::_Move_s instead.
_Move_s	Copies a specified number of characters in a sequence to another, possible overlapping, sequence.
not_eof	Tests whether a character is the end-of-file (EOF) character.
to_char_type	Converts an `int_type` character to the corresponding `char_type` character and returns the result.
to_int_type	Converts a `char_type` character to the corresponding `int_type` character and returns the result.

Requirements

Header: <string>

Namespace: std

char_traits::assign

Assigns one character value to another or to a range of elements in a string.

```
C++
```

```
static void assign(char_type& _CharTo,
    const char_type& _CharFrom);

static char_type *assign(char_type* strTo,
    size_t _Num,
    char_type _CharFrom);
```

Parameters

_ CharFrom The character whose value is to be assigned.

_CharTo
The element that is to be assigned the character value.

strTo
The string or character array whose initial elements are to be assigned character values.

_Num
The number of elements that are going to be assigned values.

Return Value

The second member function returns a pointer to the string whose first *_Num* elements have been assigned values of *_CharFrom*.

Example

```
C++
```

```
// char_traits_assign.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
```

```
    using namespace std;

    // The first member function assigning
    // one character value to another character
    char ChTo = 't';
    const char ChFrom = 'f';
    cout << "The initial characters ( ChTo , ChFrom ) are: ( "
         << ChTo << " , " << ChFrom << " )." << endl;
    char_traits<char>::assign ( ChTo , ChFrom );
    cout << "After assigning, the characters ( ChTo , ChFrom ) are: ( "
         << ChTo << " , " << ChFrom << " )." << endl << endl;

    // The second member function assigning
    // character values to initial part of a string
    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type* result1;
    cout << "The target string s1 is: " << s1 << endl;
    result1 = char_traits<char>::assign ( s1 , 4 , 'f' );
    cout << "The result1 = assign ( s1 , 4 , 'f' ) is: "
         << result1 << endl;
}
```

Output

```
The initial characters ( ChTo , ChFrom ) are: ( t , f ).
After assigning, the characters ( ChTo , ChFrom ) are: ( f , f ).

The target string s1 is: abcd-1234-abcd
The result1 = assign ( s1 , 4 , 'f' ) is: ffff-1234-abcd
```

char_traits::char_type

A type of character.

C++

```
typedef CharType char_type;
```

Remarks

The type is a synonym for the template parameter `CharType`.

Example

See the example for copy for an example of how to declare and use `char_type`.

char_traits::compare

Compares up to a specified number of characters in two strings.

```
C++
```

```
static int compare(const char_type* str1,
    const char_type* str2,
    size_t _Num);
```

Parameters

str1
The first of two strings to be compared to each other.

str2
The second of two strings to be compared to each other.

_Num
The number of elements in the strings to be compared.

Return Value

A negative value if the first string is less than the second string, 0 if the two strings are equal, or a positive value if the first string is greater than the second string.

Remarks

The comparison between the strings is made element by element, first testing for equality and then, if a pair of elements in the sequence tests not equal, they are tested for less than.

If two strings compare equal over a range but one is longer than the other, then the shorter of the two is less than the longer one.

Example

```
C++
```

```
// char_traits_compare.cpp
// compile with: /EHsc
#include <string>
#include <iostream>
```

```
int main() {
   using namespace std;

   char_traits<char>::char_type* s1 = "CAB";
   char_traits<char>::char_type* s2 = "ABC";
   char_traits<char>::char_type* s3 = "ABC";
   char_traits<char>::char_type* s4 = "ABCD";

   cout << "The string s1 is: " << s1 << endl;
   cout << "The string s2 is: " << s2 << endl;
   cout << "The string s3 is: " << s3 << endl;
   cout << "The string s4 is: " << s4 << endl;

   int comp1, comp2, comp3, comp4;
   comp1 = char_traits<char>::compare ( s1 , s2 , 2 );
   comp2 = char_traits<char>::compare ( s2 , s3 , 3 );
   comp3 = char_traits<char>::compare ( s3 , s4 , 4 );
   comp4 = char_traits<char>::compare ( s4 , s3 , 4 );
   cout << "compare ( s1 , s2 , 2 ) = " << comp1 << endl;
   cout << "compare ( s2 , s3 , 3 ) = " << comp2 << endl;
   cout << "compare ( s3 , s4 , 4 ) = " << comp3 << endl;
   cout << "compare ( s4 , s3 , 4 ) = " << comp4 << endl;
}
```

char_traits::copy

Copies a specified number of characters from one string to another.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using char_traits::_Copy_s instead.

C++

```
static char_type *copy(char_type* _To,
    const char_type* _From,
    size_t _Num);
```

Parameters

_To
The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From
The element at the beginning of the source string or character array to be copied.

_Num
The number of elements to be copied.

Return Value

The first element copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example

C++

```
// char_traits_copy.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type s1[] = "abcd-1234-abcd";
   char_traits<char>::char_type s2[] = "ABCD-1234";
   char_traits<char>::char_type* result1;
   cout << "The source string is: " << s1 << endl;
   cout << "The destination string is: " << s2 << endl;
   // Note: char_traits::copy is potentially unsafe, consider
   // using char_traits::_Copy_s instead.
   result1 = char_traits<char>::copy ( s1 , s2 , 4 );  // C4996
   cout << "The result1 = copy ( s1 , s2 , 4 ) is: "
        << result1 << endl;
}
```

Output

```
The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = copy ( s1 , s2 , 4 ) is: ABCD-1234-abcd
```

char_traits::_Copy_s

Copies a specified number of characters from one string to another.

```
C++
```

```
static char_type *_Copy_s(
    char_type* dest,
    size_t dest_size,
    const char_type* _From,
    size_t count);
```

Parameters

dest
The string or character array targeted to receive the copied sequence of characters.

dest_size
The size of *dest*. If `char_type` is **`char`**, then this size is in bytes. If `char_type` is **`wchar_t`**, then this size is in words.

_From
The source string or character array to be copied.

count
The number of elements to be copied.

Return Value

The string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination character sequences must not overlap.

Example

```
C++
```

```
// char_traits__Copy_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
```

```
    using namespace std;

    char_traits<char>::char_type s1[] = "abcd-1234-abcd";
    char_traits<char>::char_type s2[] = "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string is: " << s1 << endl;
    cout << "The destination string is: " << s2 << endl;
    result1 = char_traits<char>::_Copy_s(s1,
        char_traits<char>::length(s1), s2, 4);
    cout << "The result1 = _Copy_s(s1, "
         << "char_traits<char>::length(s1), s2, 4) is: "
         << result1 << endl;
}
```

Output

```
The source string is: abcd-1234-abcd
The destination string is: ABCD-1234
The result1 = _Copy_s(s1, char_traits<char>::length(s1), s2, 4) is: ABCD-
1234-abcd
```

char_traits::eof

Returns the end-of-file (EOF) character.

C++

```
static int_type eof();
```

Return Value

The EOF character.

Remarks

A value that represents end of file (such as EOF or WEOF).

The C++ standard states that this value must not correspond to a valid `char_type` value. The Microsoft C++ compiler enforces this constraint for type **`char`**, but not for type **`wchar_t`**. The example below demonstrates this.

Example

C++

```
// char_traits_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::char_type ch1 = 'x';
    char_traits<char>::int_type int1;
    int1 = char_traits<char>::to_int_type(ch1);
    cout << "char_type ch1 is '" << ch1 << "' and corresponds to int_type "
         << int1 << "." << endl << endl;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

Output

```
char_type ch1 is 'x' and corresponds to int_type 120.

The eof marker for char_traits<char> is: -1
The eof marker for char_traits<wchar_t> is: 65535
```

char_traits::eq

Tests whether two `char_type` characters are equal.

C++

```
static bool eq(const char_type& _Ch1, const char_type& _Ch2);
```

Parameters

_Ch1

The first of two characters to be tested for equality.

_Ch2

The second of two characters to be tested for equality.

Return Value

`true` if the first character is equal to the second character; otherwise `false`.

Example

C++

```
// char_traits_eq.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'x';

   // Testing for equality
   bool b1 = char_traits<char>::eq ( ch1 , ch2 );
   if ( b1 )
      cout << "The character ch1 is equal "
           << "to the character ch2." << endl;
   else
      cout << "The character ch1 is not equal "
           << "to the character ch2." << endl;

   // An equivalent and alternatively test procedure
   if ( ch1 == ch3 )
      cout << "The character ch1 is equal "
           << "to the character ch3." << endl;
   else
      cout << "The character ch1 is not equal "
           << "to the character ch3." << endl;
}
```

Output

```
The character ch1 is not equal to the character ch2.
The character ch1 is equal to the character ch3.
```

char_traits::eq_int_type

Tests whether two characters represented as `int_type`s are equal or not.

C++

```
static bool eq_int_type(const int_type& _Ch1, const int_type& _Ch2);
```

Parameters

_Ch1
The first of the two characters to be tested for equality as `int_type`s.

_Ch2
The second of the two characters to be tested for equality as `int_type`s.

Return Value

`true` if the first character is equal to the second character; otherwise **`false`**.

Example

C++

```
// char_traits_eq_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'x';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;
```

```
   // Testing for equality of int_type representations
   bool b1 = char_traits<char>::eq_int_type ( int1 , int2 );
   if ( b1 )
      cout << "The int_type representation of character ch1\n "
           << "is equal to the int_type representation of ch2."
           << endl;
   else
      cout << "The int_type representation of character ch1\n is "
           << "not equal to the int_type representation of ch2."
           << endl;

   // An equivalent and alternatively test procedure
   if ( int1 == int3 )
      cout << "The int_type representation of character ch1\n "
           << "is equal to the int_type representation of ch3."
           << endl;
   else
      cout << "The int_type representation of character ch1\n is "
           << "not equal to the int_type representation of ch3."
           << endl;
}
```

Output

```
The char_types and corresponding int_types are:
    ch1 = x corresponding to int1 = 120.
    ch2 = y corresponding to int1 = 121.
    ch3 = x corresponding to int1 = 120.

The int_type representation of character ch1
is not equal to the int_type representation of ch2.
The int_type representation of character ch1
is equal to the int_type representation of ch3.
```

char_traits::find

Searches for the first occurrence of a specified character in a range of characters.

C++

```
static const char_type* find(const char_type* str,
    size_t _Num,
    const char_type& _Ch);
```

Parameters

str
The first character in the string to be searched.

_Num
The number of positions, counting from the first, in the range to be searched.

_Ch
The character to be searched for in the range.

Return Value

A pointer to the first occurrence of the specified character in the range if a match is found; otherwise, a null pointer.

Example

C++

```
// char_traits_find.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   const char* s1 = "f2d-1234-abcd";
   const char* result1;
   cout << "The string to be searched is: " << s1 << endl;

   // Searching for a 'd' in the first 6 positions of string s1
   result1 = char_traits<char>::find ( s1 , 6 , 'd');
   cout << "The character searched for in s1 is: "
        << *result1 << endl;
   cout << "The string beginning with the first occurrence\n "
        << "of the character 'd' is: " << result1 << endl;

   // When no match is found the NULL value is returned
   const char* result2;
   result2 = char_traits<char>::find ( s1 , 3 , 'a');
   if ( result2 == NULL )
      cout << "The result2 of the search is NULL." << endl;
   else
      cout << "The result2 of the search  is: " << result1
           << endl;
}
```

Output

```
The string to be searched is: f2d-1234-abcd
The character searched for in s1 is: d
The string beginning with the first occurrence
of the character 'd' is: d-1234-abcd
The result2 of the search is NULL.
```

char_traits::int_type

An integer type that can represent a character of type `char_type` or an end-of-file (EOF) character.

C++

```
typedef long int_type;
```

Remarks

It must be possible to type cast a value of type `CharType` to `int_type` then back to `CharType` without altering the original value.

Example

See the example for eq_int_type for an example of how to declare and use `int_type`.

char_traits::length

Returns the length of a string.

C++

```
static size_t length(const char_type* str);
```

Parameters

str
The C-string whose length is to be measured.

Return Value

The number of elements in the sequence being measured, not including the null terminator.

Example

C++

```
// char_traits_length.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   const char* str1= "Hello";
   cout << "The C-string str1 is: " << str1 << endl;

   size_t lenStr1;
   lenStr1 = char_traits<char>::length ( str1 );
   cout << "The length of C-string str1 is: "
        << lenStr1 << "." << endl;
}
```

Output

```
The C-string str1 is: Hello
The length of C-string str1 is: 5.
```

char_traits::lt

Tests whether one character is less than another.

C++

```
static bool lt(const char_type& _Ch1, const char_type& _Ch2);
```

Parameters

_Ch1
The first of two characters to be tested for less than.

_Ch2
The second of two characters to be tested for less than.

Return Value

`true` if the first character is less than the second character; otherwise `false`.

Example

C++

```
// char_traits_lt.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::char_type ch2 =  'y';
   char_traits<char>::char_type ch3 =  'z';

   // Testing for less than
   bool b1 = char_traits<char>::lt ( ch1 , ch2 );
   if ( b1 )
      cout << "The character ch1 is less than "
           << "the character ch2." << endl;
   else
      cout << "The character ch1 is not less "
           << "than the character ch2." << endl;

   // An equivalent and alternatively test procedure
   if ( ch3 <  ch2 )
      cout << "The character ch3 is less than "
           << "the character ch2." << endl;
   else
      cout << "The character ch3 is not less "
           << "than the character ch2." << endl;
}
```

Output

```
The character ch1 is less than the character ch2.
The character ch3 is not less than the character ch2.
```

char_traits::move

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.

This method is potentially unsafe, as it relies on the caller to check that the passed values are correct. Consider using char_traits::_Move_s instead.

```
C++
```

```
static char_type *move(char_type* _To,
    const char_type* _From,
    size_t _Num);
```

Parameters

_To
The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

_From
The element at the beginning of the source string or character array to be copied.

_Num
The number of elements to be copied from the source string.

Return Value

The first element *_To* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

```
C++
```

```
// char_traits_move.cpp
// compile with: /EHsc /W3
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type sFrom1[] =  "abcd-1234-abcd";
```

```
    char_traits<char>::char_type sTo1[] =  "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string sFrom1 is: " << sFrom1 << endl;
    cout << "The destination stringsTo1 is: " << sTo1 << endl;
    // Note: char_traits::move is potentially unsafe, consider
    // using char_traits::_Move_s instead.
    result1 = char_traits<char>::move ( sTo1 ,  sFrom1 , 4 );  // C4996
    cout << "The result1 = move ( sTo1 , sFrom1 , 4 ) is: "
         << result1 << endl << endl;

    // When source and destination overlap
    char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
    char_traits<char>::char_type* result2;
    cout << "The source/destination string sToFrom2 is: "
         << sToFrom2 << endl;
    const char* findc = char_traits<char>::find ( sToFrom2 , 4 , 'c' );
    // Note: char_traits::move is potentially unsafe, consider
    // using char_traits::_Move_s instead.
    result2 = char_traits<char>::move ( sToFrom2 , findc , 8 );  // C4996
    cout << "The result2 = move ( sToFrom2 , findc , 8 ) is: "
         << result2 << endl;
}
```

Output

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = move ( sTo1 , sFrom1 , 4 ) is: abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = move ( sToFrom2 , findc , 8 ) is: cd-1234-4-ABCD
```

char_traits::_Move_s

Copies a specified number of characters in a sequence to another, possibly overlapping sequence.

C++

```
static char_type *_Move_s(
    char_type* dest,
    size_t dest_size,
    const char_type* _From,
    size_t count);
```

Parameters

dest
The element at the beginning of the string or character array targeted to receive the copied sequence of characters.

dest_size
The size of *dest*. If `char_type` is **`char`**, then this is in bytes. If `char_type` is **`wchar_t`**, then this is in words.

_From
The element at the beginning of the source string or character array to be copied.

count
The number of elements to be copied from the source string.

Return Value

The first element *dest* copied into the string or character array targeted to receive the copied sequence of characters.

Remarks

The source and destination may overlap.

Example

C++

```
// char_traits__Move_s.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
    using namespace std;

    char_traits<char>::char_type sFrom1[] =  "abcd-1234-abcd";
    char_traits<char>::char_type sTo1[] =  "ABCD-1234";
    char_traits<char>::char_type* result1;
    cout << "The source string sFrom1 is: " << sFrom1 << endl;
    cout << "The destination stringsTo1 is: " << sTo1 << endl;
    result1 = char_traits<char>::_Move_s(sTo1,
        char_traits<char>::length(sTo1), sFrom1, 4);
    cout << "The result1 = _Move_s(sTo1, "
         << "char_traits<char>::length(sTo1), sFrom1, 4) is: "
         << result1 << endl << endl;
```

```
    // When source and destination overlap
    char_traits<char>::char_type sToFrom2[] = "abcd-1234-ABCD";
    char_traits<char>::char_type* result2;
    cout << "The source/destination string sToFrom2 is: "
         << sToFrom2 << endl;
    const char* findc = char_traits<char>::find(sToFrom2, 4, 'c');
    result2 = char_traits<char>::_Move_s(sToFrom2,
        char_traits<char>::length(sToFrom2), findc, 8);
    cout << "The result2 = _Move_s(sToFrom2, "
        << "char_traits<char>::length(sToFrom2), findc, 8) is: "
         << result2 << endl;
}
```

Output

```
The source string sFrom1 is: abcd-1234-abcd
The destination stringsTo1 is: ABCD-1234
The result1 = _Move_s(sTo1, char_traits<char>::length(sTo1), sFrom1, 4) is:
abcd-1234

The source/destination string sToFrom2 is: abcd-1234-ABCD
The result2 = _Move_s(sToFrom2, char_traits<char>::length(sToFrom2), findc,
8) is: cd-1234-4-ABCD
```

char_traits::not_eof

Tests whether a character is not the end-of-file (EOF) character or is the EOF.

C++

```
static int_type not_eof(const int_type& _Ch);
```

Parameters

_Ch
The character represented as an `int_type` to be tested for whether it is the EOF character or not.

Return Value

The `int_type` representation of the character tested, if the `int_type` of the character is not equal to that of the EOF character.

If the character `int_type` value is equal to the EOF `int_type` value, then **`false`**.

Example

C++

```
// char_traits_not_eof.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( ) {
   using namespace std;

   char_traits<char>::char_type ch1 =  'x';
   char_traits<char>::int_type int1;
   int1 = char_traits<char>:: to_int_type ( ch1 );
   cout << "The char_type ch1 is " << ch1
        << " corresponding to int_type: "
        << int1 << "." << endl;

   // EOF member function
   char_traits <char>::int_type int2 = char_traits<char>::eof ( );
   cout << "The eofReturn is: " << int2 << endl;

   // Testing for EOF or another character
   char_traits <char>::int_type eofTest1, eofTest2;
   eofTest1 = char_traits<char>::not_eof ( int1 );
   if ( !eofTest1 )
      cout << "The eofTest1 indicates ch1 is an EOF character."
              << endl;
   else
      cout << "The eofTest1 returns: " << eofTest1
           << ", which is the character: "
           <<  char_traits<char>::to_char_type ( eofTest1 )
           << "." << endl;

   eofTest2 = char_traits<char>::not_eof ( int2 );
   if ( !eofTest2 )
      cout << "The eofTest2 indicates int2 is an EOF character."
           << endl;
   else
      cout << "The eofTest1 returns: " << eofTest2
           << ", which is the character: "
           <<  char_traits<char>::to_char_type ( eofTest2 )
           << "." << endl;
}
```

Output

```
The char_type ch1 is x corresponding to int_type: 120.
The eofReturn is: -1
The eofTest1 returns: 120, which is the character: x.
The eofTest2 indicates int2 is an EOF character.
```

char_traits::off_type

An integer type that can represent offsets between positions in a stream.

```
C++
```

```
typedef streamoff off_type;
```

Remarks

The type is a signed integer that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for streamoff, but it has essentially the same properties as that type.

char_traits::pos_type

An integer type that can represent positions in a stream.

```
C++
```

```
typedef streampos pos_type;
```

Remarks

The type describes an object that can store all the information needed to restore an arbitrary file-position indicator within a stream. It is typically a synonym for streampos, but in any case it has essentially the same properties as that type.

char_traits::state_type

A type that represents the conversion state for multibyte characters in a stream.

```
C++
```

```
typedef implementation-defined state_type;
```

Remarks

The type describes an object that can represent a conversion state. It is typically a synonym for `mbstate_t`, but in any case it has essentially the same properties as that type.

char_traits::to_char_type

Converts an `int_type` character to the corresponding `char_type` character and returns the result.

```
C++
```

```
static char_type to_char_type(const int_type& _Ch);
```

Parameters

_Ch
The `int_type` character to be represented as a `char_type`.

Return Value

The `char_type` character corresponding to the `int_type` character.

A value of *_Ch* that cannot be represented as such yields an unspecified result.

Remarks

The conversion operations to_int_type and `to_char_type` are inverse to each other, so that:

`to_int_type` (`to_char_type` (*x*)) == *x*

for any `int_type` *x* and

`to_char_type` (`to_int_type` (*x*)) == *x*

for any `char_type` *x*.

Example

```
C++
```

```cpp
// char_traits_to_char_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;

   char_traits<char>::char_type ch1 =  'a';
   char_traits<char>::char_type ch2 =  'b';
   char_traits<char>::char_type ch3 =  'a';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

   // Converting from int_type back to char_type
   char_traits<char>::char_type rec_ch1;
   rec_ch1 = char_traits<char>:: to_char_type ( int1);
   char_traits<char>::char_type rec_ch2;
   rec_ch2 = char_traits<char>:: to_char_type ( int2);

   cout << "The recovered char_types and corresponding int_types are:"
        << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
        << int1 << "."
        << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
        << int2 << "." << endl << endl;

   // Testing that the conversions are inverse operations
   bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
   if ( b1 )
      cout << "The recovered char_type of ch1"
           << " is equal to the original ch1." << endl;
   else
      cout << "The recovered char_type of ch1"
           << " is not equal to the original ch1." << endl;

   // An equivalent and alternatively test procedure
   if ( rec_ch2 == ch2 )
      cout << "The recovered char_type of ch2"
           << " is equal to the original ch2." << endl;
   else
      cout << "The recovered char_type of ch2"
```

```
            << " is not equal to the original ch2." << endl;
}
```

Output

```
The char_types and corresponding int_types are:
    ch1 = a corresponding to int1 = 97.
    ch2 = b corresponding to int1 = 98.
    ch3 = a corresponding to int1 = 97.

The recovered char_types and corresponding int_types are:
    recovered ch1 = a from int1 = 97.
    recovered ch2 = b from int2 = 98.

The recovered char_type of ch1 is equal to the original ch1.
The recovered char_type of ch2 is equal to the original ch2.
```

char_traits::to_int_type

Converts a `char_type` character to the corresponding `int_type` character and returns the result.

C++

```
static int_type to_int_type(const char_type& _Ch);
```

Parameters

_Ch
The `char_type` character to be represented as an `int_type`.

Return Value

The `int_type` character corresponding to the `char_type` character.

Remarks

The conversion operations `to_int_type` and to_char_type are inverse to each other, so that:

`to_int_type` (`to_char_type` (*x*)) == *x*

for any `int_type` *x*, and

`to_char_type` (`to_int_type` (*x*)) == *x*

for any `char_type` *x*.

Example

C++

```
// char_traits_to_int_type.cpp
// compile with: /EHsc
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   char_traits<char>::char_type ch1 = 'a';
   char_traits<char>::char_type ch2 = 'b';
   char_traits<char>::char_type ch3 = 'a';

   // Converting from char_type to int_type
   char_traits<char>::int_type int1, int2 , int3;
   int1 =char_traits<char>:: to_int_type ( ch1 );
   int2 =char_traits<char>:: to_int_type ( ch2 );
   int3 =char_traits<char>:: to_int_type ( ch3 );

   cout << "The char_types and corresponding int_types are:"
        << "\n    ch1 = " << ch1 << " corresponding to int1 = "
        << int1 << "."
        << "\n    ch2 = " << ch2 << " corresponding to int1 = "
        << int2 << "."
        << "\n    ch3 = " << ch3 << " corresponding to int1 = "
        << int3 << "." << endl << endl;

   // Converting from int_type back to char_type
   char_traits<char>::char_type rec_ch1;
   rec_ch1 = char_traits<char>:: to_char_type ( int1);
   char_traits<char>::char_type rec_ch2;
   rec_ch2 = char_traits<char>:: to_char_type ( int2);

   cout << "The recovered char_types and corresponding int_types are:"
        << "\n    recovered ch1 = " << rec_ch1 << " from int1 = "
        << int1 << "."
        << "\n    recovered ch2 = " << rec_ch2 << " from int2 = "
        << int2 << "." << endl << endl;

   // Testing that the conversions are inverse operations
   bool b1 = char_traits<char>::eq ( rec_ch1 , ch1 );
   if ( b1 )
      cout << "The recovered char_type of ch1"
           << " is equal to the original ch1." << endl;
   else
```

```
      cout << "The recovered char_type of ch1"
           << " is not equal to the original ch1." << endl;

   // An equivalent and alternatively test procedure
   if ( rec_ch2 == ch2 )
      cout << "The recovered char_type of ch2"
           << " is equal to the original ch2." << endl;
   else
      cout << "The recovered char_type of ch2"
           << " is not equal to the original ch2." << endl;
}
```

Output

```
The char_types and corresponding int_types are:
    ch1 = a corresponding to int1 = 97.
    ch2 = b corresponding to int1 = 98.
    ch3 = a corresponding to int1 = 97.

The recovered char_types and corresponding int_types are:
    recovered ch1 = a from int1 = 97.
    recovered ch2 = b from int2 = 98.

The recovered char_type of ch1 is equal to the original ch1.
The recovered char_type of ch2 is equal to the original ch2.
```

See also

Thread Safety in the C++ Standard Library

`char_traits<char>` Struct

Article • 12/06/2021

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type `char`.

Syntax

```
C++

template <>
struct char_traits<char>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type `char`.

Example

See the typedefs and member functions of the class template char_traits Class

`char_traits<char16_t>` Struct

Article • 12/06/2021

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type `char16_t`.

Syntax

```
C++

template <>
struct char_traits<char16_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of the type `char16_t`.

Requirements

Header: <string>

Namespace: std

See also

<string>
char_traits Struct
Thread Safety in the C++ Standard Library

`char_traits<char32_t>` Struct

Article • 12/06/2021

A struct that is a specialization of the template struct **char_traits<CharType>** to an element of type `char32_t`.

Syntax

```
C++

template <>
struct char_traits<char32_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type `char32_t`.

Requirements

Header: <string>

Namespace: std

See also

<string>
char_traits Struct

`char_traits<wchar_t>` Struct

Article • 12/06/2021

A class that is a specialization of the template struct **char_traits<CharType>** to an element of type `wchar_t`.

Syntax

```
C++

template <>
struct char_traits<wchar_t>;
```

Remarks

Specialization allows the struct to take advantage of library functions that manipulate objects of this type `wchar_t`.

Requirements

Header: <string>

Namespace: std

See also

char_traits Struct
Thread Safety in the C++ Standard Library

<string_view>

Article • 03/01/2022

Defines the class template `basic_string_view` and related types and operators. (Requires compiler option std:c++17 or later.)

Syntax

```
C++

#include <string_view>
```

Remarks

The `string_view` family of template specializations provides an efficient way to pass a read-only, exception-safe, non-owning handle to the character data of any string-like objects with the first element of the sequence at position zero. A function parameter of type `string_view` (which is a typedef for `basic_string_view<char>`) can accept arguments such as `std::string`, `char*`, or any other string-like class of narrow characters for which an implicit conversion to `string_view` is defined. Similarly, a parameter of `wstring_view`, `u16string_view` or `u32string_view` can accept any string type for which an implicit conversion is defined. For more information, see basic_string_view Class.

Typedefs

Type name	Description
string_view	A specialization of the class template `basic_string_view` with elements of type `char`.
wstring_view	A specialization of the class template `basic_string_view` with elements of type `wchar_t`.
u16string_view	A specialization of the class template `basic_string_view` with elements of type `char16_t`.
u32string_view	A specialization of the class template `basic_string_view` with elements of type `char32_t`.

Operators

The `<string_view>` operators can compare `string_view` objects to objects of any convertible string types.

Operator	Description
operator!=	Tests if the object on the left side of the operator isn't equal to the object on the right side.
operator==	Tests if the object on the left side of the operator is equal to the object on the right side.
operator<	Tests if the object on the left side of the operator is less than to the object on the right side.
operator<=	Tests if the object on the left side of the operator is less than or equal to the object on the right side.
operator<\<	A template function that inserts a `string_view` into an output stream.
operator>	Tests if the object on the left side of the operator is greater than to the object on the right side.
operator>=	Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

Literals

Operator	Description
sv	Constructs a `string_view`, `wstring_view`, `u16string_view`, or `u32string_view` depending on the type of the string literal to which it's appended.

Classes

Class	Description
basic_string_view Class	A class template that provides a read-only view into a sequence of arbitrary character-like objects.
hash	Function object that produces a hash value for a `string_view`.

Requirements

- **Header:** `<string_view>`
- **Namespace:** `std`
- **Compiler Option:** std:c++17 or later.

See also

Header Files Reference

`<string_view>` operators

Article • 11/07/2022

Use these operators to compare two `string_view` objects, or a `string_view` and some other string object (for example std::string, or `char*`) for which an implicit conversion is provided.

operator!=
operator>
operator>=
operator<
operator<<
operator<=
operator==
operator""sv

`operator!=`

Tests if the object on the left side of the operator is not equal to the object on the right side.

C++

```
template <class CharType, class Traits>
bool operator!=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator!=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator!=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator isn't lexicographically equal to the object on the right side; otherwise **`false`**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the `string_view` on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. If they have the same number of elements and the elements are all equal, the two objects are equal. Otherwise, they're unequal.

`operator==`

Tests if the object on the left side of the operator is equal to the object on the right side.

C++

```
template <class CharType, class Traits>
bool operator==(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator==(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator==(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left
Any convertible string type or an object of type `basic_string_view` to be compared.

right
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator is lexicographically equal to the object on the right side; otherwise **`false`**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the `string_view` on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. If they have the same number of elements and the elements are all equal, the two objects are equal.

`operator<`

Tests if the object on the left side of the operator is less than the object on the right side.

C++

```
template <class CharType, class Traits>
bool operator<(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator<(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator<(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left
Any convertible string type or an object of type `basic_string_view` to be compared.

right
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator is lexicographically less than the object on the right side; otherwise **`false`**.

Remarks

An implicit conversion must exist from *convertible_string_type* to the string_view on the other side.

The comparison is based on a pairwise lexicographical comparison of the character sequences. When the first unequal pair of characters is encountered, the result of that comparison is returned. If no unequal characters are found, but one sequence is shorter, the shorter sequence is less than the longer one. In other words, "cat" is less than "cats".

Example

C++

```
#include <string>
#include <string_view>

using namespace std;

int main()
{
    string_view sv1 { "ABA" };
    string_view sv2{ "ABAC" };
    string_view sv3{ "ABAD" };
    string_view sv4{ "ABACE" };

    bool result = sv2 > sv1; // true
    result = sv3 > sv2; // true
    result = sv3 != sv1; // true
    result = sv4 < sv3; // true because `C` < `D`
}
```

`operator<=`

Tests if the object on the left side of the operator is less than or equal to the object on the right side.

C++

```
template <class CharType, class Traits>
bool operator<=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator<=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator<=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left
Any convertible string type or an object of type `basic_string_view` to be compared.

right
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator is lexicographically less than or equal to the object on the right side; otherwise `false`.

Remarks

See operator<.

`operator<<`

Writes a `string_view` into an output stream.

C++

```
template <class CharType, class Traits>
inline basic_ostream<CharType, Traits>& operator<<(
    basic_ostream<CharType, Traits>& Ostr, const basic_string_view<CharType,
Traits> Str);
```

Parameters

Ostr

an output stream being written to.

Str

The string_view to be entered into an output stream.

Return Value

an output stream being written to.

Remarks

Use this operator to insert the contents of a `string_view` into an output stream, for example using std::cout.

`operator>`

Tests if the object on the left side of the operator is greater than the object on the right side.

C++

```
template <class CharType, class Traits>
bool operator>(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator>(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator>(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

left

Any convertible string type or an object of type `basic_string_view` to be compared.

right
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator is lexicographically greater than the `string_view` object on the right side; otherwise **`false`**.

Remarks

See operator<.

`operator>=`

Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

```
C++
```

```
template <class CharType, class Traits>
bool operator>=(
    const basic_string_view<CharType, Traits>& left,
    const basic_string_view<CharType, Traits>& right);

template <class CharType, class Traits>
bool operator>=(
    const basic_string_view<CharType, Traits>& left,
    convertible_string_type right);

template <class CharType, class Traits>
bool operator>=(
    convertible_string_type left,
    const basic_string_view<CharType, Traits>& right);
```

Parameters

`left`
Any convertible string type or an object of type `basic_string_view` to be compared.

`right`
Any convertible string type or an object of type `basic_string_view` to be compared.

Return Value

`true` if the object on the left side of the operator is lexicographically greater than or equal to the object on the right side; otherwise `false`.

Remarks

See operator<.

`operator"" sv` (`string_view` literal)

Constructs a `string_view` from a string literal. Requires namespace `std::literals::string_view_literals`.

Example

C++

```
using namespace std;
using namespace literals::string_view_literals;

    string_view sv{ "Hello"sv };
    wstring_view wsv{ L"Hello"sv };
    u16string_view sv16{ u"Hello"sv };
    u32string_view sv32{ U"Hello"sv };
```

Requirements

/std:c++17 or later.

See also

<string_view>

`<string_view>` typedefs

Article • 12/07/2021

string_view
u16string_view
u32string_view
wstring_view

string_view

A type that describes a specialization of the class template basic_string_view with elements of type `char`.

```
C++

typedef basic_string_view<char, char_traits<char>> string_view;
```

Remarks

The following are equivalent declarations:

```
C++

string_view str("Hello");

basic_string_view<char> str("Hello");
```

For a list of string constructors, see basic_string::basic_string.

u16string_view

A type that describes a specialization of the class template basic_string_view with elements of type `char16_t`.

```
C++

typedef basic_string_view<char16_t, char_traits<char16_t>> u16string_view;
```

Remarks

For a list of string constructors, see basic_string::basic_string.

u32string_view

A type that describes a specialization of the class template basic_string_view with elements of type `char32_t`.

```
C++

typedef basic_string_view<char32_t, char_traits<char32_t>> u32string_view;
```

Remarks

For a list of string constructors, see basic_string::basic_string.

wstring_view

A type that describes a specialization of the class template basic_string_view with elements of type `wchar_t`.

```
C++

typedef basic_string_view<wchar_t, char_traits<wchar_t>> wstring_view;
```

Remarks

The following are equivalent declarations:

```
C++

wstring_view wstr(L"Hello");

basic_string_view<wchar_t> wstr(L"Hello");
```

For a list of string constructors, see basic_string::basic_string.

> ⓘ Note
>
> The size of `wchar_t` is two bytes on Windows but this is not necessarily the case for all platforms. If you need a string_view wide character type with a width that is

guaranteed to remain the same on all platforms, use **u16string_view** or **u32string_view**.

See also

<string_view>

`basic_string_view` class

Article • 10/25/2022

The class template `basic_string_view<charT>` was added in C++17 to serve as a safe and efficient way for a function to accept various unrelated string types without the function having to be templatized on those types. The class holds a non-owning pointer to a contiguous sequence of character data, and a length that specifies the number of characters in the sequence. No assumption is made about whether the sequence is null-terminated.

The standard library defines several specializations based on the type of the elements:

- `string_view`
- `wstring_view`
- `u16string_view`
- `u32string_view`

A `basic_string_view` describes the minimum common interface necessary to read string data. It provides const access to the underlying data; it makes no copies (except for the `copy` function). The data may or may not contain null values (`\0`) at any position. A `basic_string_view` has no control over the object's lifetime. It's the caller's responsibility to ensure that the underlying string data is valid.

A function that accepts a parameter of type `string_view` can be made to work with any string-like type, without making the function into a template, or constraining the function to a particular subset of string types. The only requirement is that an implicit conversion exists from the string type to `string_view`. All the standard string types are implicitly convertible to a `string_view` that contains the same element type. In other words, a `std::string` is convertible to a `string_view` but not to a `wstring_view`.

The following example shows a non-template function `f` that takes a parameter of type `wstring_view`. It can be called with arguments of type `std::wstring`, `wchar_t*`, and `winrt::hstring`.

C++

```
// compile with: /std:c++17
// string_view that uses elements of wchar_t
void f(wstring_view);

// pass a std::wstring:
const std::wstring& s { L"Hello" };
```

```
f(s);

// pass a C-style null-terminated string (string_view is not null-
terminated):
const wchar_t* ns = L"Hello";
f(ns);

// pass a C-style character array of len characters (excluding null
terminator):
const wchar_t* cs { L"Hello" };
size_t len { 5 };
f({cs,len});

// pass a WinRT string
winrt::hstring hs { L"Hello" };
f(hs);
```

Syntax

C++

```
template <class CharType, class Traits = char_traits<CharType>>
class basic_string_view;
```

Parameters

CharType

The type of the characters that are stored in the `basic_string_view`. The C++ Standard Library provides the following typedefs for specializations of this template.

- string_view for elements of type **`char`**
- wstring_view, for **`wchar_t`**
- u16string_view for **`char16_t`**
- u32string_view for **`char32_t`**.

Traits

Defaults to char_traits<*CharType*>.

Constructors

Constructor	Description
basic_string_view	Constructs a `basic_string_view` that is empty or else points to all or part of some other string object's data, or to a C-style character array.

Typedefs

Type name	Description
`const_iterator`	Random-access iterator that can read `const` elements.
`const_pointer`	`using const_pointer = const value_type*;`
`const_reference`	`using const_reference = const value_type&;`
`const_reverse_iterator`	`using const_reverse_iterator = std::reverse_iterator<const_iterator>;`
`difference_type`	`using difference_type = ptrdiff_t;`
`iterator`	`using iterator = const_iterator;`
`npos`	`static constexpr size_type npos = size_type(-1);`
`pointer`	`using pointer = value_type*;`
`reference`	`using reference = value_type&;`
`reverse_iterator`	`using reverse_iterator = const_reverse_iterator;`
`size_type`	`using size_type = size_t;`
`traits_type`	`using traits_type = Traits;`
`value_type`	`using value_type = CharType;`

Member operators

Operator	Description
operator=	Assigns a `basic_string_view` or convertible string object to another `basic_string_view`.
operator[]	Returns the element at the specified index.

Member functions

Member function	Description
at	Returns a `const_reference` to the element at a specified location.

Member function	Description
back	Returns a `const_reference` to the last element.
begin	Returns a `const` iterator addressing the first element. (`basic_string_view`s are immutable.)
cbegin	Same as begin.
cend	Returns a `const` iterator that points to one past the last element.
copy	Copies at most a specified number of characters from an indexed position in a source `basic_string_view` to a target character array. (Not recommended. Use `_Copy_s` instead.)
_Copy_s	Secure CRT copy function.
compare	Compares a `basic_string_view` with a specified `basic_string_view` to determine if they're equal or if one is lexicographically less than the other.
crbegin	Same as rbegin.
crend	Same as rend.
data	Returns a raw non-owning pointer to the character sequence.
empty	Tests whether the `basic_string_view` contains characters.
end	Same as cend.
ends_with C++20	Check whether a string view ends with a specified suffix.
find	Searches in a forward direction for the first occurrence of a substring that matches a specified sequence of characters.
find_first_not_of	Searches for the first character that isn't any element of a specified `basic_string_view` or convertible string object.
find_first_of	Searches for the first character that matches any element of a specified `basic_string_view` or convertible string object.
find_last_not_of	Searches for the last character that isn't any element of a specified `basic_string_view` or convertible string object.
find_last_of	Searches for the last character that is an element of a specified `basic_string_view` or convertible string object.
front	Returns a `const_reference` to the first element.
length	Returns the current number of elements.

Member function	Description
max_size	Returns the maximum number of characters a `basic_string_view` could contain.
rbegin	Returns a `const` iterator that addresses the first element in a reversed `basic_string_view`.
remove_prefix	Moves the pointer forward by the specified number of elements.
remove_suffix	Reduces the size of the view by the specified number of elements starting from the back.
rend	Returns a `const` iterator that points to one past the last element in a reversed `basic_string_view`.
rfind	Searches a `basic_string_view` in reverse for the first occurrence of a substring that matches a specified sequence of characters.
size	Returns the current number of elements.
starts_with C++20	Check whether a string view starts with a given prefix.
substr	Returns a substring of a specified length starting at a specified index.
swap	Exchange the contents of two `basic_string_view` objects.

Remarks

If a function is asked to generate a sequence longer than max_size elements, the function reports a length error by throwing an object of type length_error.

Requirements

std:c++17 or later.

Header: `<string_view>`

Namespace: `std`

`basic_string_view::at`

Returns a `const_reference` to the character at the specified zero-based index.

```
C++
```

```
constexpr const_reference at(size_type offset) const;
```

Parameters

offset
The index of the element to be referenced.

Return value

A `const_reference` to the character at the position specified by the parameter index.

Remarks

The first element has an index of zero and the following elements are indexed consecutively by the positive integers, so that a `basic_string_view` of length *n* has an *`n`*th element indexed by the number `n - 1`. **`at`** throws an exception for invalid indices, unlike operator[].

In general, we recommend that **`at`** for sequences such as `std::vector` and `basic_string_view` should never be used. An invalid index passed to a sequence is a logic error that should be discovered and fixed during development. If a program isn't certain that its indices are valid, it should test them, not call `at()` and rely on exceptions to defend against careless programming.

For more information, see basic_string_view::operator[].

Example

C++

```
// basic_string_view_at.cpp
// compile with: /EHsc
#include <string_view>
#include <iostream>

int main()
{
    using namespace std;

    const string_view  str1("Hello world");
    string_view::const_reference refStr2 = str1.at(8); // 'r'
}
```

`basic_string_view::back`

Returns a `const_reference` to the last element.

```
C++
```

```
constexpr const_reference back() const;
```

Return value

A `const_reference` to the last element in the `basic_string_view`.

Remarks

Throws an exception if the `basic_string_view` is empty.

Keep in mind that after a `basic_string_view` is modified, for example by calling `remove_suffix`, then the element returned by this function is no longer the last element in the underlying data.

Example

A `string_view` that is constructed with a C string literal doesn't include the terminating null. So in the following example, `back` returns `'p'` and not `'\0'`.

```
C++
```

```
char c[] = "Help"; // char[5]
string_view sv{ c };
cout << sv.size(); // size() == 4
cout << sv.back() << endl; // p
```

Embedded nulls are treated as any other character:

```
C++
```

```
string_view e = "embedded\0nulls"sv;
cout << boolalpha << (e.back() == 's'); // true
```

`basic_string_view::basic_string_view`

Constructs a `basic_string_view`.

```
C++
```

```
constexpr basic_string_view() noexcept;
constexpr basic_string_view(const basic_string_view&) noexcept = default;
constexpr basic_string_view(const charT* str);
constexpr basic_string_view(const charT* str, size_type len);
```

Parameters

`str`

The pointer to the character values.

`len`

The number of characters to include in the view.

Remarks

The constructors with a `charT*` parameter assume that the input is null-terminated, but the terminating null isn't included in the `basic_string_view`.

You can also construct a `basic_string_view` with a literal. See operator"" sv.

`basic_string_view::begin`

Same as cbegin.

```
C++
```

```
constexpr const_iterator begin() const noexcept;
```

Return value

Returns a `const_iterator` addressing the first element.

`basic_string_view::cbegin`

Returns a `const_iterator` that addresses the first element in the range.

```
C++
```

```
constexpr const_iterator cbegin() const noexcept;
```

Return value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

`basic_string_view::cend`

Returns a `const_iterator` that addresses the location just beyond the last element in a range.

C++

```
constexpr const_iterator cend() const noexcept;
```

Return value

A `const` random-access iterator that points just beyond the end of the range.

Remarks

The value returned by `cend` shouldn't be dereferenced.

`basic_string_view::compare`

Does a case-sensitive comparison with a specified `basic_string_view` (or a convertible string type) to determine if the two objects are equal or if one is lexicographically less than the other. The <string_view> operators use this member function to do comparisons.

C++

```
constexpr int compare(basic_string_view strv) const noexcept;
constexpr int compare(size_type pos, size_type num, basic_string_view strv)
const;
constexpr int compare(size_type pos, size_type num, basic_string_view strv,
size_type offset, size_type num2) const;
constexpr int compare(const charT* ptr) const;
```

```
constexpr int compare(size_type pos, size_type num, const charT* ptr) const;
constexpr int compare(size_type pos, size_type num, const charT* ptr,
size_type num2) const;
```

Parameters

strv

The `basic_string_view` that is to be compared to this `basic_string_view`.

pos

The index of this `basic_string_view` at which the comparison begins.

num

The maximum number of characters from this `basic_string_view` to be compared.

num2

The maximum number of characters from *strv* to be compared.

offset

The index of *strv* at which the comparison begins.

ptr

The C string to be compared to this `basic_string_view`.

Return value

- A negative value if this `basic_string_view` is less than *strv* or *ptr*
- Zero if the two character sequences are equal
- A positive value if this `basic_string_view` is greater than *strv* or *ptr*

Remarks

The `compare` member functions do a case-sensitive comparison of either all or part of each character sequence.

Example

C++

```
// basic_string_view_compare.cpp
// compile with: /EHsc
#include <string_view>
```

```cpp
#include <iostream>
#include <string>

using namespace std;

string to_alpha(int result)
{
   if (result < 0) return " less than ";
   else if (result == 0) return " equal to ";
   else return " greater than ";
}

int main()
{
   // The first member function compares
   // two string_views
   string_view sv_A("CAB");
   string_view sv_B("CAB");
   cout << "sv_A is " << sv_A << endl;
   cout << "sv_B is " << sv_B << endl;
   int comp1 = sv_A.compare(sv_B);
   cout << "sv_A is" << to_alpha(comp1) << "sv_B.\n";

   // The second member function compares part of
   // an operand string_view to another string_view
   string_view sv_C("AACAB");
   string_view sv_D("CAB");
   cout << "sv_C is: " << sv_C << endl;
   cout << "sv_D is: " << sv_D << endl;
   int comp2a = sv_C.compare(2, 3, sv_D);
   cout << "The last three characters of sv_C are"
        << to_alpha(comp2a) << "sv_D.\n";

   int comp2b = sv_C.compare(0, 3, sv_D);
   cout << "The first three characters of sv_C are"
        << to_alpha(comp2b) << "sv_D.\n";

   // The third member function compares part of
   // an operand string_view to part of another string_view
   string_view sv_E("AACAB");
   string_view sv_F("DCABD");
   cout << "sv_E: " << sv_E << endl;
   cout << "sv_F is: " << sv_F << endl;
   int comp3a = sv_E.compare(2, 3, sv_F, 1, 3);
   cout << "The three characters from position 2 of sv_E are"
        << to_alpha(comp3a)
        << "the 3 characters of sv_F from position 1.\n";

   // The fourth member function compares
   // an operand string_view to a C string
   string_view sv_G("ABC");
   const char* cs_A = "DEF";
   cout << "sv_G is: " << sv_G << endl;
   cout << "cs_A is: " << cs_A << endl;
   int comp4a = sv_G.compare(cs_A);
```

```
    cout << "sv_G is" << to_alpha(comp4a) << "cs_A.\n";

    // The fifth member function compares part of
    // an operand string_view to a C string
    string_view sv_H("AACAB");
    const char* cs_B = "CAB";
    cout << "sv_H is: " << sv_H << endl;
    cout << "cs_B is: " << cs_B << endl;
    int comp5a = sv_H.compare(2, 3, cs_B);
    cout << "The last three characters of sv_H are"
       << to_alpha(comp5a) << "cs_B.\n";

    // The sixth member function compares part of
    // an operand string_view to part of an equal length of
    // a C string
    string_view sv_I("AACAB");
    const char* cs_C = "ACAB";
    cout << "sv_I is: " << sv_I << endl;
    cout << "cs_C: " << cs_C << endl;
    int comp6a = sv_I.compare(1, 3, cs_C, 3);
    cout << "The 3 characters from position 1 of sv_I are"
       << to_alpha(comp6a) << "the first 3 characters of cs_C.\n";
}
```

Output

```
sv_A is CAB
sv_B is CAB
sv_A is equal to sv_B.
sv_C is: AACAB
sv_D is: CAB
The last three characters of sv_C are equal to sv_D.
The first three characters of sv_C are less than sv_D.
sv_E: AACAB
sv_F is: DCABD
The three characters from position 2 of sv_E are equal to the 3 characters
of sv_F from position 1.
sv_G is: ABC
cs_A is: DEF
sv_G is less than cs_A.
sv_H is: AACAB
cs_B is: CAB
The last three characters of sv_H are equal to cs_B.
sv_I is: AACAB
cs_C: ACAB
The 3 characters from position 1 of sv_I are equal to the first 3 characters
of cs_C.
```

basic_string_view::copy

Copies at most a specified number of characters from an indexed position in a source `basic_string_view` to a target character array. We recommend that you use the secure function basic_string_view::_Copy_s instead.

```
C++
```

```
size_type copy(charT* ptr, size_type count, size_type offset = 0) const;
```

Parameters

`ptr`
The target character array to which the elements are to be copied.

`count`
The number of characters to be copied, at most, from the source `basic_string_view`.

`offset`
The beginning position in the source `basic_string_view` from which copies are to be made.

Return value

The number of characters copied.

Remarks

A null character isn't appended to the end of the copy.

`basic_string_view::_Copy_s`

Secure CRT copy function to be used instead of copy.

```
C++
```

```
size_type _Copy_s(
    value_type* dest,
    size_type dest_size,
    size_type count,
    size_type _Off = 0) const;
```

Parameters

dest

The target character array to which the elements are to be copied.

dest_size

The size of *dest*.

count The number of characters to be copied, at most, from the source string.

_Off

The beginning position in the source string from which copies are to be made.

Return value

The number of characters copied.

Remarks

A null character isn't appended to the end of the copy.

For more information, see c-runtime-library/security-features-in-the-crt.

basic_string_view::crbegin

Returns a `const_reverse_iterator` that addresses the first element in a reversed `basic_string_view`.

```
C++
```

```
constexpr const_reverse_iterator crbegin() const noexcept;
```

Return value

A `const_reverse_iterator` that addresses the first element in a reversed `basic_string_view`.

basic_string_view::crend

Same as rend.

```
C++
```

```
constexpr const_reverse_iterator crend() const noexcept;
```

Return value

Returns a `const_reverse_iterator` that addresses one past the end of a reversed `basic_string_view`.

`basic_string_view::data`

Returns a raw non-owning pointer to the const character sequence of the object that was used to construct the `basic_string_view`.

C++

```
constexpr value_type *data() const noexcept;
```

Return value

A pointer-to-const to the first element of the character sequence.

Remarks

The pointer can't modify the characters.

A sequence of `basic_string_view` characters isn't necessarily null-terminated. The return type for `data` isn't a valid C string, because no null character gets appended. The null character `\0` has no special meaning in an object of type `basic_string_view` and may be a part of the `basic_string_view` object just like any other character.

`basic_string_view::empty`

Tests whether the `basic_string_view` contains characters or not.

C++

```
constexpr bool empty() const noexcept;
```

Return value

`true` if the `basic_string_view` object contains no characters; `false` if it has at least one character.

Remarks

The member function is equivalent to size() == 0.

`basic_string_view::end`

Returns a random-access `const_iterator` that points to one past the last element.

```
C++
```

```
constexpr const_iterator end() const noexcept;
```

Return value

Returns a random-access `const_iterator` that points to one past the last element.

Remarks

`end` is used to test whether a `const_iterator` has reached the end of its `basic_string_view`. The value returned by `end` shouldn't be dereferenced.

`basic_string_view::ends_with`

Check whether the string view ends with the specified suffix.

```
C++
```

```
bool ends_with(const CharType c) const noexcept;
bool ends_with(const CharType* const x) const noexcept;
bool ends_with(const basic_string_view sv) const noexcept;
```

Parameters

c

The single character suffix to look for.

`sv`

A string view containing the suffix to look for.
You can pass a `std::basic_string`, which converts to a `basic_string_view`.

`x`

Null-terminated character string containing the suffix to look for.

Return value

`true` if the string view ends with the specified suffix; `false` otherwise.

Remarks

`ends_with()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

See starts_with to check if a string view starts with the specified prefix.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <string>
#include <iostream>

int main()
{
    std::cout << std::boolalpha; // so booleans show as 'true'/'false'
    std::cout << std::string_view("abcdefg").ends_with('g') << '\n';
    std::cout << std::string_view("abcdefg").ends_with("eFg") << '\n';

    std::basic_string<char> str2 = "efg";
    std::cout << std::string_view("abcdefg").ends_with(str2);

    return 0;
}
```

Output

```
true
false
true
```

`basic_string_view::find`

Searches a `basic_string_view` in a forward direction for the first occurrence of a character or substring that matches a specified sequence of character(s).

```
C++
```

```
constexpr size_type find(basic_string_view str, size_type offset = 0) const
noexcept;
constexpr size_type find(charT chVal, size_type offset = 0) const noexcept;
constexpr size_type find(const charT* ptr, size_type offset, size_type
count) const;
constexpr size_type find(const charT* ptr, size_type offset = 0) const;
```

Parameters

`str`

The `basic_string_view` for which the member function is to search.

`chVal`

The character value for which the member function is to search.

`offset`

Index at which the search is to begin.

`ptr`

The C string for which the member function is to search.

`count`

The number of characters in *`ptr`*, counting forward from the first character.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

`basic_string_view::find_first_not_of`

Searches for the first character that isn't an element of a specified `basic_string_view` or convertible string object.

```
C++
```

```
constexpr size_type find_first_not_of(basic_string_view str, size_type
offset = 0) const noexcept;
```

```
constexpr size_type find_first_not_of(charT chVal, size_type offset = 0)
const noexcept;
constexpr size_type find_first_not_of(const charT* ptr, size_type offset,
size_type count) const;
constexpr size_type find_first_not_of(const charT* ptr, size_type offset =
0) const;
```

Parameters

str

The `basic_string_view` for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

`basic_string_view::find_first_of`

Searches for the first character that matches any element of a specified `basic_string_view`.

C++

```
constexpr size_type find_first_of(basic_string_view str, size_type offset =
0) const noexcept;
constexpr size_type find_first_of(charT chVal, size_type offset = 0) const
noexcept;
constexpr size_type find_first_of(const charT* str, size_type offset,
size_type count) const;
```

```
constexpr size_type find_first_of(const charT* str, size_type offset = 0)
const;
```

Parameters

chVal

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

str

The `basic_string_view` for which the member function is to search.

Return value

The index of the first character of the substring searched for when successful; otherwise `npos`.

`basic_string_view::find_last_not_of`

Searches for the last character that isn't any element of a specified `basic_string_view`.

C++

```
constexpr size_type find_last_not_of(basic_string_view str, size_type offset
= npos) const noexcept;
constexpr size_type find_last_not_of(charT chVal, size_type offset = npos)
const noexcept;
constexpr size_type find_last_not_of(const charT* ptr, size_type offset,
size_type count) const;
constexpr size_type find_last_not_of(const charT* ptr, size_type offset =
npos) const;
```

Parameters

str

The `basic_string_view` for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to finish.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in *ptr*.

Return value

The index of the first character of the substring searched for when successful; otherwise `string_view::npos`.

`basic_string_view::find_last_of`

Searches for the last character that matches any element of a specified `basic_string_view`.

C++

```
constexpr size_type find_last_of(basic_string_view str, size_type offset =
npos) const noexcept;
constexpr size_type find_last_of(charT chVal, size_type offset = npos) const
noexcept;
constexpr size_type find_last_of(const charT* ptr, size_type offset,
size_type count) const;
constexpr size_type find_last_of(const charT* ptr, size_type offset = npos)
const;
```

Parameters

str

The `basic_string_view` for which the member function is to search.

chVal

The character value for which the member function is to search.

offset

Index at which the search is to finish.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

Return value

The index of the last character of the substring searched for when successful; otherwise `npos`.

`basic_string_view::front`

Returns a `const_reference` to the first element.

```
C++
```

```
constexpr const_reference front() const;
```

Return value

A `const_reference` to the first element.

Remarks

Throws an exception if the `basic_string_view` is empty.

`basic_string_view::length`

Returns the current number of elements.

```
C++
```

```
constexpr size_type length() const noexcept;
```

Remarks

The member function is the same as size.

`basic_string_view::max_size`

Returns the maximum number of characters a `basic_string_view` can contain.

```
C++
```

```
constexpr size_type max_size() const noexcept;
```

Return value

The maximum number of characters a `basic_string_view` can contain.

Remarks

An exception of type length_error is thrown when an operation produces a `basic_string_view` with a length greater than `max_size()`.

`basic_string_view::operator=`

Assigns a `basic_string_view` or convertible string object to another `basic_string_view`.

```
C++
```

```
constexpr basic_string_view& operator=(const basic_string_view&) noexcept =
default;
```

Example

```
C++
```

```
   string_view s = "Hello";
   string_view s2 = s;
```

`basic_string_view::operator[]`

Provides a `const_reference` to the character with a specified index.

```
C++
```

```
constexpr const_reference operator[](size_type offset) const;
```

Parameters

`offset`
The index of the element to be referenced.

Return value

A `const_reference` to the character at the position specified by the parameter index.

Remarks

The first element has an index of zero, and the following elements are indexed consecutively by the positive integers, so that a `basic_string_view` of length *`n`* has an *`n`*th element indexed by the number `n-1`.

`operator[]` is faster than the member function at for providing read access to the elements of a `basic_string_view`.

`operator[]` doesn't check whether the index passed as an argument is valid. An invalid index passed to `operator[]` results in undefined behavior.

The reference returned may be invalidated if the underlying string data is modified or deleted by the owning object.

When compiling with _ITERATOR_DEBUG_LEVEL set to 1 or 2, a runtime error will occur if you attempt to access an element outside the bounds of the `basic_string_view`. For more information, see Checked Iterators.

`basic_string_view::rbegin`

Returns a `const` iterator to the first element in a reversed `basic_string_view`.

```
C++
```

```
constexpr const_reverse_iterator rbegin() const noexcept;
```

Return value

Returns a random-access iterator to the first element in a reversed `basic_string_view`, addressing what would be the last element in the corresponding unreversed `basic_string_view`.

Remarks

`rbegin` is used with a reversed `basic_string_view` just as begin is used with a `basic_string_view`. `rbegin` can be used to initialize an iteration backwards.

`basic_string_view::remove_prefix`

Moves the pointer forward by the specified number of elements.

```
C++
```

```
constexpr void remove_prefix(size_type n);
```

Remarks

Leaves the underlying data unchanged. Moves the `basic_string_view` pointer forward by `n` elements and sets the private `size` data member to `size - n`.

`basic_string_view::remove_suffix`

Reduces the size of the view by the specified number of elements starting from the back.

```
C++
```

```
constexpr void remove_suffix(size_type n);
```

Remarks

Leaves the underlying data and the pointer to it unchanged. Sets the private `size` data member to `size - n`.

`basic_string_view::rend`

Returns a `const` iterator that points to one past the last element in a reversed `basic_string_view`.

```
C++
```

```
constexpr reverse_iterator rend() const noexcept;
```

Return value

A `const` reverse random-access iterator that points to one past the last element in a reversed `basic_string_view`.

Remarks

`rend` is used with a reversed `basic_string_view` just as end is used with a `basic_string_view`. `rend` can be used to test whether a reverse iterator has reached the end of its `basic_string_view`. The value returned by `rend` shouldn't be dereferenced.

`basic_string_view::rfind`

Searches a `basic_string_view` in reverse for a substring that matches a specified sequence of characters.

```
C++
```

```
constexpr size_type rfind(basic_string_view str, size_type offset = npos)
const noexcept;
constexpr size_type rfind(charT chVal, size_type offset = npos) const
noexcept;
constexpr size_type rfind(const charT* ptr, size_type offset, size_type
count) const;
constexpr size_type rfind(const charT* ptr, size_type offset = npos) const;
```

Parameters

`chVal`

The character value for which the member function is to search.

offset

Index at which the search is to begin.

ptr

The C string for which the member function is to search.

count

The number of characters, counting forward from the first character, in the C string for which the member function is to search.

str

The `basic_string_view` for which the member function is to search.

Return value

The index of the first character of the substring when successful; otherwise `npos`.

`basic_string_view::size`

Returns the number of elements in the `basic_string_view`.

```
C++
```

```
constexpr size_type size() const noexcept;
```

Return value

The length of the `basic_string_view`.

Remarks

A `basic_string_view` can modify its length, for example by `remove_prefix` and `remove_suffix`. Because this doesn't modify the underlying string data, the size of a `basic_string_view` isn't necessarily the size of the underlying data.

`basic_string_view::starts_with`

Check whether the string view starts with the specified prefix.

```
C++
```

```
bool starts_with(const CharType c) const noexcept;
bool starts_with(const CharType* const x) const noexcept;
bool starts_with(const basic_string_view sv) const noexcept;
```

Parameters

`c`

The single character prefix to look for.

`sv`

A string view containing the prefix to look for.

You can pass a `std::basic_string`, which converts to a string view.

`x`

Null-terminated character string containing the prefix to look for.

Return value

`true` if the string starts with the specified prefix; `false` otherwise.

Remarks

`starts_with()` is new in C++20. To use it, specify the std:c++20 or later compiler option.

See ends_with to see if a string ends with a suffix.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <string>
#include <iostream>

int main()
{
    std::cout << std::boolalpha; // so booleans show as 'true'/'false'
    std::cout << std::string_view("abcdefg").starts_with('b') << '\n';
    std::cout << std::string_view("abcdefg").starts_with("aBc") << '\n';

    std::basic_string<char> str2 = "abc";
    std::cout << std::string_view("abcdefg").starts_with(str2);

    return 0;
}
```

Output

```
false
false
true
```

`basic_string_view::substr`

Returns a `basic_string_view` that represents (at most) the specified number of characters from a specified position.

C++

```
constexpr basic_string_view substr(size_type offset = 0, size_type count =
npos) const;
```

Parameters

`offset`
An index locating the element at the position from which the copy is made, with a default value of 0.

`count`
The number of characters to include in the substring, if they're present.

Return value

A `basic_string_view` object that represents the specified subsequence of elements.

`basic_string_view::swap`

Exchanges two `basic_string_view`s, in other words the pointers to the underlying string data, and the size values.

C++

```
constexpr void swap(basic_string_view& sv) noexcept;
```

Parameters

sv

The source `basic_string_view` whose pointer and size values are to be exchanged with that of the destination `basic_string_view`.

See also

<string_view>

Thread Safety in the C++ Standard Library

`hash<string_view>` Specialization

Article • 12/06/2021

A template specialization that produces a hash value given a string_view.

```
C++
```

```
template <class CharType, class Traits>
struct hash<basic_string_view<CharType, Traits>>
{
    typedef basic_string_view<CharType, Traits> argument_type;
    typedef size_t result_type;

    size_t operator()(const basic_string_view<CharType, Traits>) const
        noexcept;
};
```

Remarks

The hash of a string_view equals the hash of the underlying string object.

Example

```
C++
```

```
//compile with: /std:c++17
#include <string>
#include <string_view>
#include <iostream>

using namespace std;

int main()
{
    string_view sv{ "Hello world" };
    string s{ "Hello world" };
    cout << boolalpha << (hash<string_view>{}(sv)
        == hash<string>{}(s)); // true
}
```

<strstream>

Article • 12/07/2021

Defines several classes that support iostreams operations on sequences stored in an allocated array of `char` object. Such sequences are easily converted to and from C strings.

Requirements

Header: <strstream>

Namespace: std

Remarks

Objects of type `strstream` work with `char` *, which are C strings. Use <sstream> to work with objects of type basic_string.

> **Note**
>
> The classes in <strstream> are deprecated. Consider using the classes in <sstream> instead.

Members

Classes

Name	Description
strstreambuf Class	The class describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a `char` array object.
istrstream Class	The class describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf.
ostrstream Class	The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class strstreambuf.
strstream Class	The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf.

Functions

```
C++
```

```
void freeze(bool freezefl = true);
char* str();
int pcount();
```

See also

<strstream>

Header Files Reference

Thread Safety in the C++ Standard Library

iostream Programming

iostreams Conventions

strstreambuf Class

Article • 08/03/2021

Describes a stream buffer that controls the transmission of elements to and from a sequence of elements stored in a `char` array object.

Syntax

```
C++
```

```
class strstreambuf : public streambuf
```

Remarks

Depending on how the object is constructed, it can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `strstreambuf` stores several bits of mode information as its `strstreambuf` mode. These bits indicate whether the controlled sequence:

- Has been allocated and needs to be freed eventually.
- Is modifiable.
- Is extendable by reallocating storage.
- Has been frozen and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object.

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control `strstreambuf` allocation. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

> **Note**
>
> This class is deprecated. Consider using **stringbuf** or **wstringbuf** instead.

Constructors

Constructor	Description
strstreambuf	Constructs an object of type `strstreambuf`.

Member functions

Member function	Description
freeze	Causes a stream buffer to be unavailable through stream buffer operations.
overflow	A protected virtual function that can be called when a new character is inserted into a full buffer.
pbackfail	A protected virtual member function that tries to put back an element into the input stream, and then make it the current element (pointed to by the next pointer).
pcount	Returns a count of the number of elements written to the controlled sequence.
seekoff	A protected virtual member function that tries to alter the current positions for the controlled streams.
seekpos	A protected virtual member function that tries to alter the current positions for the controlled streams.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.
underflow	A protected virtual function to extract the current element from the input stream.

Requirements

Header: <strstream>

Namespace: std

strstreambuf::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

C++

```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit
A `bool` indicating whether you want the stream to be frozen.

Remarks

If *_Freezeit* is true, the function alters the stored `strstreambuf` mode to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

str implies `freeze`.

> **Note**
>
> A frozen buffer will not be freed during `strstreambuf` destruction. You must unfreeze the buffer before it is freed to avoid a memory leak.

Example

C++

```
// strstreambuf_freeze.cpp
// compile with: /EHsc

#include <iostream>
#include <strstream>

using namespace std;

void report(strstream &x)
{
    if (!x.good())
        cout << "stream bad" << endl;
    else
        cout << "stream good" << endl;
}

int main()
{
    strstream x;

    x << "test1";
    cout << "before freeze: ";
    report(x);

    // Calling str freezes stream.
    cout.write(x.rdbuf()->str(), 5) << endl;
```

```
    cout << "after freeze: ";
    report(x);

    // Stream is bad now, wrote on frozen stream
    x << "test1.5";
    cout << "after write to frozen stream: ";
    report(x);

    // Unfreeze stream, but it is still bad
    x.rdbuf()->freeze(false);
    cout << "after unfreezing stream: ";
    report(x);

    // Clear stream
    x.clear();
    cout << "after clearing stream: ";
    report(x);

    x << "test3";
    cout.write(x.rdbuf()->str(), 10) << endl;

    // Clean up.  Failure to unfreeze stream will cause a
    // memory leak.
    x.rdbuf()->freeze(false);
}
```

Output

```
before freeze: stream good
test1
after freeze: stream good
after write to frozen stream: stream bad
after unfreezing stream: stream bad
after clearing stream: stream good
test1test3
```

strstreambuf::overflow

A protected virtual function that can be called when a new character is inserted into a full buffer.

C++

```
virtual int overflow(int _Meta = EOF);
```

Parameters

_Meta
The character to insert into the buffer, or `EOF`.

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, if *_Meta* == `EOF`, it returns some value other than `EOF`. Otherwise, it returns *_Meta*.

Remarks

If *_Meta* != `EOF`, the protected virtual member function tries to insert the element `(char)_Meta` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored strstreambuf mode says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. Extending the output buffer this way also extends any associated input buffer.

strstreambuf::pbackfail

A protected virtual member function that tries to put back an element into the input stream, and then makes it the current element (pointed to by the next pointer).

```
C++
```

```
virtual int pbackfail(int _Meta = EOF);
```

Parameters

_Meta
The character to insert into the buffer, or `EOF`.

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, if *_Meta* == `EOF`, it returns some value other than `EOF`. Otherwise, it returns *_Meta*.

Remarks

The protected virtual member function tries to put back an element into the input buffer, and then make it the current element (pointed to by the next pointer).

If *_Meta* == `EOF`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `ch = (char)_Meta`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `ch`, it can decrement the next pointer for the input buffer.
- If a putback position is available, and if the strstreambuf mode says the controlled sequence is modifiable, the function can store `ch` into the putback position and decrement the next pointer for the input buffer.

strstreambuf::pcount

Returns a count of the number of elements written to the controlled sequence.

```
C++
```

```
streamsize pcount() const;
```

Return Value

A count of the number of elements written to the controlled sequence.

Remarks

Specifically, if pptr is a null pointer, the function returns zero. Otherwise, it returns `pptr - pbase`.

Example

```
C++
```

```
// strstreambuf_pcount.cpp
// compile with: /EHsc
#include <iostream>
#include <strstream>
using namespace std;
```

```
int main( )
{
   strstream x;
   x << "test1";
   cout << x.rdbuf( )->pcount( ) << endl;
   x << "test2";
   cout << x.rdbuf( )->pcount( ) << endl;
}
```

strstreambuf::seekoff

A protected virtual member function that tries to alter the current positions for the controlled streams.

C++

```
virtual streampos seekoff(streamoff _Off,
    ios_base::seekdir _Way,
    ios_base::openmode _Which = ios_base::in | ios_base::out);
```

Parameters

_Off
The position to seek for relative to *_Way*.

_Way
The starting point for offset operations. See seekdir for possible values.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class strstreambuf, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `_Way == ios_base::beg`, the new position is the beginning of the stream plus *_Off*.
- If `_Way == ios_base::cur`, the new position is the current stream position plus *_Off*.
- If `_Way == ios_base::end`, the new position is the end of the stream plus *_Off*.

If `_Which & ios_base::in` is nonzero and the input buffer exist, the function alters the next position to read in the input buffer. If `_Which & ios_base::out` is also nonzero, `_Way != ios_base::cur`, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if `_Which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

strstreambuf::seekpos

A protected virtual member function that tries to alter the current positions for the controlled streams.

```
C++
```

```
virtual streampos seekpos(streampos _Sp, ios_base::openmode _Which =
ios_base::in | ios_base::out);
```

Parameters

_Sp
The position to seek for.

_Which
Specifies the mode for the pointer position. The default is to allow you to modify the read and write positions.

Return Value

If the function succeeds in altering either or both stream positions, it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position. To

determine if the stream position is invalid, compare the return value with `pos_type(off_type(-1))`.

Remarks

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class strstreambuf, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by *_Sp*.

If `_Which` & **ios_base::in** is nonzero and the input buffer exists, the function alters the next position to read in the input buffer. If `_Which` & `ios_base::out` is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read. Otherwise, if `_Which` & `ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

strstreambuf::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
C++
```

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

No terminating null element exists, unless you explicitly insert one.

Example

See strstreambuf::freeze for a sample that uses **str**.

strstreambuf::strstreambuf

Constructs an object of type `strstreambuf`.

```
C++
```

```
explicit strstreambuf(streamsize count = 0);

strstreambuf(void (* alloc_func)(size_t),
    void (* free_func)(void*));

strstreambuf(char* getptr,
    streamsize count,
    char* putptr = 0);

strstreambuf(signed char* getptr,
    streamsize count,
    signed char* putptr = 0);

strstreambuf(unsigned char* getptr,
    streamsize count,
    unsigned char* putptr = 0);

strstreambuf(const char* getptr,
    streamsize count);

strstreambuf(const signed char* getptr,
    streamsize count);

strstreambuf(const unsigned char* getptr,
    streamsize count);
```

Parameters

alloc_func
The function used to allocate buffer memory.

count
Determines the length of the buffer pointed to by *getptr*. If *getptr* is not an argument (first constructor form), a suggested allocation size for the buffers.

_Freefunc
The function used to free buffer memory.

getptr
A buffer used for input.

putptr
A buffer used for output.

Remarks

The first constructor stores a null pointer in all the pointers controlling the input buffer, the output buffer, and strstreambuf allocation. It sets the stored strstreambuf mode to make the controlled sequence modifiable and extendable. It also accepts *count* as a suggested initial allocation size.

The second constructor behaves like the first, except that it stores *alloc_func* as the pointer to the function to call to allocate storage and *free_func* as the pointer to the function to call to free that storage.

The three constructors:

```
C++

strstreambuf(char *getptr,
    streamsize count,
    char *putptr = 0);

strstreambuf(signed char *getptr,
    streamsize count,
    signed char *putptr = 0);

strstreambuf(unsigned char *getptr,
    streamsize count,
    unsigned char *putptr = 0);
```

also behave like the first, except that *getptr* designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements *N* in the array is determined as follows:

- If (*count* > 0), then *N* is *count*.
- If (*count* == 0), then *N* is `strlen((const char *) getptr )`.
- If (*count* < 0), then *N* is **INT_MAX**.

If *putptr* is a null pointer, the function establishes just an input buffer by executing:

```
C++

setg(getptr,
    getptr,
    getptr + N);
```

Otherwise, it establishes both input and output buffers by executing:

```
C++

setg(getptr,
    getptr,
    putptr);

setp(putptr,
    getptr + N);
```

In this case, *putptr* must be in the interval [*getptr*, *getptr* + *N*].

Finally, the three constructors:

```
C++

strstreambuf(const char *getptr,
    streamsize count);

strstreambuf(const signed char *getptr,
    streamsize count);

strstreambuf(const unsigned char *getptr,
    streamsize count);
```

all behave the same as:

```
C++

streambuf((char *)getptr, count);
```

except that the stored mode makes the controlled sequence neither modifiable nor extendable.

strstreambuf::underflow

A protected virtual function to extract the current element from the input stream.

```
C++

virtual int underflow();
```

Return Value

If the function cannot succeed, it returns `EOF`. Otherwise, it returns the current element in the input stream, converted as described above.

Remarks

The protected virtual member function endeavors to extract the current element `ch` from the input buffer, then advance the current stream position, and return the element as `(int)(unsigned char)ch`. It can do so in only one way: if a read position is available, it takes `ch` as the element stored in the read position and advances the next pointer for the input buffer.

See also

streambuf
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

istrstream Class

Article • 08/03/2021

Describes an object that controls extraction of elements and encoded objects from a stream buffer of class strstreambuf.

Syntax

```
C++
```

```
class istrstream : public istream
```

Remarks

The object stores an object of class `strstreambuf`.

> **Note**
>
> This class is deprecated. Consider using **istringstream** or **wistringstream** instead.

Constructors

Constructor	Description
istrstream	Constructs an object of type `istrstream`.

Member functions

Member function	Description
rdbuf	Returns a pointer to the stream's associated `strstreambuf` object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: <strstream>

Namespace: std

istrstream::istrstream

Constructs an object of type `istrstream`.

```
C++
```

```
explicit istrstream(
    const char* ptr);

explicit istrstream(
    char* ptr);

istrstream(
    const char* ptr,
    streamsize count);

istrstream(
    char* ptr,
    int count);
```

Parameters

count
The length of the buffer (*ptr*).

ptr
The contents with which the buffer is initialized.

Remarks

All the constructors initialize the base class by calling istream(**sb**), where `sb` is the stored object of class strstreambuf. The first two constructors also initialize `sb` by calling `strstreambuf( ( const char *) ptr, 0 )`. The remaining two constructors instead call `strstreambuf( ( const char *) ptr, count )`.

istrstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
C++
```

```
strstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses `rdbuf`.

istrstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
C++
```

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstream::str for a sample that uses `str`.

See also

istream
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

ostrstream Class

Article • 08/03/2021

Describes an object that controls insertion of elements and encoded objects into a stream buffer of class strstreambuf.

Syntax

```
C++
```

```
class ostrstream : public ostream
```

Remarks

The object stores an object of class `strstreambuf`.

> **Note**
>
> This class is deprecated. Consider using **ostringstream** or **wostringstream** instead.

Constructors

Constructor	Description
ostrstream	Constructs an object of type `ostrstream`.

Member functions

Member function	Description
freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
rdbuf	Returns a pointer to the stream's associated `strstreambuf` object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: <strstream>

Namespace: std

ostrstream::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
C++
```

```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit
A `bool` indicating whether you want the stream to be frozen.

Remarks

The member function calls rdbuf -> freeze(_ *Freezeit*).

Example

See strstream::freeze for an example that uses `freeze`.

ostrstream::ostrstream

Constructs an object of type `ostrstream`.

```
C++
```

```
ostrstream();

ostrstream(char* ptr,
    streamsize count,
    ios_base::openmode _Mode = ios_base::out);
```

Parameters

ptr
The buffer.

count
The size of the buffer in bytes.

_Mode
The input and output mode of the buffer. See ios_base::openmode for more information.

Remarks

Both constructors initialize the base class by calling ostream(**sb**), where `sb` is the stored object of class strstreambuf. The first constructor also initializes `sb` by calling `strstreambuf`. The second constructor initializes the base class one of two ways:

- If `_Mode` & **ios_base::app**== 0, then `ptr` must designate the first element of an array of `count` elements, and the constructor calls `strstreambuf`(`ptr`, `count`, `ptr`).
- Otherwise, `ptr` must designate the first element of an array of count elements that contains a C string whose first element is designated by `ptr`, and the constructor calls `strstreambuf`(`ptr`, `count`, `ptr` + `strlen`(`ptr`)).

ostrstream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
C++
```

```
streamsize pcount() const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns rdbuf -> pcount.

Example

See strstream::pcount for a sample that uses `pcount`.

ostrstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
C++
```

```
strstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses `rdbuf`.

ostrstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
C++
```

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstream::str for a sample that uses `str`.

See also

ostream
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

strstream Class

Article • 08/03/2021

Describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf.

Syntax

```
C++
```

```
class strstream : public iostream
```

Remarks

The object stores an object of class `strstreambuf`.

> **Note**
>
> This class is deprecated. Consider using **stringstream** or **wstringstream** instead.

Constructors

Constructor	Description
strstream	Constructs an object of type `strstream`.

Member functions

Member function	Description
freeze	Causes a stream buffer to be unavailable through stream buffer operations.
pcount	Returns a count of the number of elements written to the controlled sequence.
rdbuf	Returns a pointer to the stream's associated `strstreambuf` object.
str	Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

Requirements

Header: <strstream>

Namespace: std

strstream::freeze

Causes a stream buffer to be unavailable through stream buffer operations.

```
C++
```

```
void freeze(bool _Freezeit = true);
```

Parameters

_Freezeit
A `bool` indicating whether you want the stream to be frozen.

Remarks

The member function calls rdbuf -> freeze(_ *Freezeit*).

Example

See strstreambuf::freeze for an example that uses `freeze`.

strstream::pcount

Returns a count of the number of elements written to the controlled sequence.

```
C++
```

```
streamsize pcount() const;
```

Return Value

The number of elements written to the controlled sequence.

Remarks

The member function returns rdbuf -> pcount.

Example

See strstreambuf::pcount for a sample of using pcount.

strstream::rdbuf

Returns a pointer to the stream's associated strstreambuf object.

```
C++
```

```
strstreambuf *rdbuf() const
```

Return Value

A pointer to the stream's associated strstreambuf object.

Remarks

The member function returns the address of the stored stream buffer of type `pointer` to strstreambuf.

Example

See strstreambuf::pcount for a sample that uses `rdbuf`.

strstream::str

Calls freeze, and then returns a pointer to the beginning of the controlled sequence.

```
C++
```

```
char *str();
```

Return Value

A pointer to the beginning of the controlled sequence.

Remarks

The member function returns rdbuf -> str.

Example

See strstreambuf::str for a sample that uses `str`.

strstream::strstream

Constructs an object of type `strstream`.

```
C++

strstream();

strstream(char* ptr,
    streamsize count,
    ios_base::openmode _Mode = ios_base::in | ios_base::out);
```

Parameters

count
The size of the buffer.

_Mode
The input and output mode of the buffer. See ios_base::openmode for more information.

ptr
The buffer.

Remarks

Both constructors initialize the base class by calling streambuf(**sb**), where `sb` is the stored object of class strstreambuf. The first constructor also initializes `sb` by calling strstreambuf. The second constructor initializes the base class one of two ways:

- If `_Mode` & **ios_base::app**== 0, then *ptr* must designate the first element of an array of `count` elements, and the constructor calls `strstreambuf`(`ptr`, `count`, `ptr`).

- Otherwise, *ptr* must designate the first element of an array of count elements that contains a C string whose first element is designated by *ptr*, and the constructor calls `strstreambuf( ptr, count, ptr + strlen( ptr) )`.

See also

iostream
Thread Safety in the C++ Standard Library
iostream Programming
iostreams Conventions

<system_error>

Article • 12/07/2021

Include the header <system_error> to define the exception class `system_error` and related templates for processing low-level system errors.

Requirements

Header: <system_error>

Namespace: std

Members

Objects

Name	Description
generic_category	Represents the category for generic errors.
is_error_code_enum_v	
is_error_condition_enum_v	
system_category	Represents the category for errors caused by low-level system overflows.

Functions

Name	Description
make_error_code	Creates an `error_code` object.
make_error_condition	Creates an `error_condition` object.

Operators

Name	Description
operator==	Tests if the object on the left side of the operator is equal to the object on the right side.

Name	Description
operator!=	Tests if the object on the left side of the operator is not equal to the object on the right side.
operator<	Tests if an object is less than the object passed in for comparison.
operator<<	

Enums

Name	Description
errc	Provides symbolic names for all the error-code macros defined by POSIX in `<errno.h>`.

Classes and Structs

Name	Description
error_category	Represents the abstract, common base for objects that describes a category of error codes.
error_code	Represents low-level system errors that are implementation-specific.
error_condition	Represents user-defined error codes.
hash	
is_error_code_enum	Represents a type predicate that tests for the error_code Class enumeration.
is_error_condition_enum	Represents a type predicate that tests for the error_condition Class enumeration.
system_error	Represents the base class for all exceptions thrown to report a low-level system overflow.

See also

Header Files Reference

`<system_error>` functions

Article • 12/07/2021

generic_category

Represents the category for generic errors.

```
C++
```

```
const error_category& generic_category() noexcept;
```

Remarks

The `generic_category` object is an implementation of error_category.

is_error_code_enum_v

```
C++
```

```
template <class T>
    inline constexpr bool is_error_code_enum_v =
is_error_code_enum<T>::value;
```

is_error_condition_enum_v

```
C++
```

```
template <class T>
    inline constexpr bool is_error_condition_enum_v =
is_error_condition_enum<T>::value;
```

make_error_code

Creates an error code object.

```
C++
```

```
error_code make_error_code(std::errc error) noexcept;
```

Parameters

error
The `std::errc` enumeration value to store in the error code object.

Return Value

The error code object.

Remarks

make_error_condition

Creates an error condition object.

```
C++
```

```
error_condition make_error_condition(std::errc error) noexcept;
```

Parameters

error
The `std::errc` enumeration value to store in the error code object.

Return Value

The error condition object.

Remarks

system_category

Represents the category for errors caused by low-level system overflows.

```
C++
```

```
const error_category& system_category() noexcept;
```

Remarks

The `system_category` object is an implementation of error_category.

`<system_error>` operators

Article • 12/07/2021

operator==

Tests if the object on the left side of the operator is equal to the object on the right side.

```
C++
```

```
bool operator==(const error_code& left,
    const error_condition& right);

bool operator==(const error_condition& left,
    const error_code& right);

bool operator==(const error_condition& left,
    const error_condition& right);
```

Parameters

left
The object to be tested for equality.

right
The object to be tested for equality.

Return Value

`true` if the objects are equal; `false` if objects are not equal.

Remarks

This function returns `left.category() == right.category() && left.value() == right.value()`.

operator!=

Tests if the object on the left side of the operator is not equal to the object on the right side.

```
```

C++

```
bool operator!=(const error_code& left, const error_condition& right);
bool operator!=(const error_condition& left, const error_code& right);
bool operator!=(const error_code& left, const error_code& right);
bool operator!=(const error_condition& left, const error_condition& right);
```

Parameters

left
The object to be tested for inequality.

right
The object to be tested for inequality.

Return Value

`true` if the object passed in *left* is not equal to the object passed in *right*; otherwise `false`.

Remarks

This function returns `!(left == right)`.

`operator<`

Tests if an object is less than the object passed in for comparison.

C++

```
template <class _Enum>
inline bool operator<(
    _Enum left,
    typename enable_if<is_error_code_enum<_Enum>::value,
    const error_code&>::type right);

template <class _Enum>
inline bool operator<(
    typename enable_if<is_error_code_enum<_Enum>::value,
    const error_code&>::type left, _Enum right);

template <class _Enum>
inline bool operator<(
    _Enum left,
    typename enable_if<is_error_condition_enum<_Enum>::value,
```

```
    const error_condition&>::type right);

template <class _Enum>
inline bool operator<(
    typename enable_if<is_error_condition_enum<_Enum>::value,
    const error_condition&>::type left, _Enum right);
```

Parameters

left
The object to be compared.

right
The object to be compared.

Return Value

`true` if the object passed in *left* is less than the object passed in *right*; Otherwise, `false`.

Remarks

This function tests the error order.

`operator<<`

C++

```
template <class charT, class traits>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&
os, const error_code& ec);
```

`<system_error>` enums

Article • 12/07/2021

errc

Provides symbolic names for all the error-code macros defined by POSIX in `<errno.h>`.

C++

```
class errc {
   address_family_not_supported = EAFNOSUPPORT,
   address_in_use = EADDRINUSE,
   address_not_available = EADDRNOTAVAIL,
   already_connected = EISCONN,
   argument_list_too_long = E2BIG,
   argument_out_of_domain = EDOM,
   bad_address = EFAULT,
   bad_file_descriptor = EBADF,
   bad_message = EBADMSG,
   broken_pipe = EPIPE,
   connection_aborted = ECONNABORTED,
   connection_already_in_progress = EALREADY,
   connection_refused = ECONNREFUSED,
   connection_reset = ECONNRESET,
   cross_device_link = EXDEV,
   destination_address_required = EDESTADDRREQ,
   device_or_resource_busy = EBUSY,
   directory_not_empty = ENOTEMPTY,
   executable_format_error = ENOEXEC,
   file_exists = EEXIST,
   file_too_large = EFBIG,
   filename_too_long = ENAMETOOLONG,
   function_not_supported = ENOSYS,
   host_unreachable = EHOSTUNREACH,
   identifier_removed = EIDRM,
   illegal_byte_sequence = EILSEQ,
   inappropriate_io_control_operation = ENOTTY,
   interrupted = EINTR,
   invalid_argument = EINVAL,
   invalid_seek = ESPIPE,
   io_error = EIO,
   is_a_directory = EISDIR,
   message_size = EMSGSIZE,
   network_down = ENETDOWN,
   network_reset = ENETRESET,
   network_unreachable = ENETUNREACH,
   no_buffer_space = ENOBUFS,
   no_child_process = ECHILD,
   no_link = ENOLINK,
   no_lock_available = ENOLCK,
```

```
    no_message_available = ENODATA,
    no_message = ENOMSG,
    no_protocol_option = ENOPROTOOPT,
    no_space_on_device = ENOSPC,
    no_stream_resources = ENOSR,
    no_such_device_or_address = ENXIO,
    no_such_device = ENODEV,
    no_such_file_or_directory = ENOENT,
    no_such_process = ESRCH,
    not_a_directory = ENOTDIR,
    not_a_socket = ENOTSOCK,
    not_a_stream = ENOSTR,
    not_connected = ENOTCONN,
    not_enough_memory = ENOMEM,
    not_supported = ENOTSUP,
    operation_canceled = ECANCELED,
    operation_in_progress = EINPROGRESS,
    operation_not_permitted = EPERM,
    operation_not_supported = EOPNOTSUPP,
    operation_would_block = EWOULDBLOCK,
    owner_dead = EOWNERDEAD,
    permission_denied = EACCES,
    protocol_error = EPROTO,
    protocol_not_supported = EPROTONOSUPPORT,
    read_only_file_system = EROFS,
    resource_deadlock_would_occur = EDEADLK,
    resource_unavailable_try_again = EAGAIN,
    result_out_of_range = ERANGE,
    state_not_recoverable = ENOTRECOVERABLE,
    stream_timeout = ETIME,
    text_file_busy = ETXTBSY,
    timed_out = ETIMEDOUT,
    too_many_files_open_in_system = ENFILE,
    too_many_files_open = EMFILE,
    too_many_links = EMLINK,
    too_many_synbolic_link_levels = ELOOP,
    value_too_large = EOVERFLOW,
    wrong_protocol_type = EPROTOTYPE,
};
```

Remarks

io_errc

Provides symbolic names for the error conditions in <iostream>. Can be used to create error_condition objects to be compared with the value that's returned by the ios_base::failure `code()` function.

```
C++
```

```
class io_errc {
   stream = 1
};
```

Remarks

Both std::make_error_code() and std::make_error_condition() are overloaded for this enum.

`ios_base::failure` can contain categories of error codes other than `error_condition`.

Example

C++

```
// io_errc.cpp
// cl.exe /nologo /W4 /EHsc /MTd

#include <iostream>

using namespace std;

int main()
{
    cin.exceptions(ios::failbit | ios::badbit);

    try {
        cin.rdbuf(nullptr); // throws io_errc::stream
    }
    catch (ios::failure& e) {
        cerr << "ios failure caught: ";
        if (e.code() == make_error_condition(io_errc::stream)) {
            cerr << "io_errc stream error condition" << endl;
        }
        else {
            cerr << "unmatched error condition code " << e.code() << endl;
        }
    }
}
```

error_category Class

Article • 06/17/2022

Represents the abstract, common base for objects that describes a category of error codes.

Syntax

C++

```
class error_category;

constexpr error_category() noexcept;
virtual ~error_category();
error_category(const error_category&) = delete
```

Remarks

Two predefined objects implement `error_category`: generic_category and system_category.

Members

Typedefs

Name	Description
value_type	A type that represents the stored error code value.

Functions

Name	Description
default_error_condition	Stores the error code value for an error condition object.
equivalent	Returns a value that specifies whether error objects are equivalent.
generic_category	
message	Returns the name of the specified error code.

Name	Description
name	Returns the name of the category.
system_category	

Operators

Name	Description
operator=	Assignment operator.
operator==	Tests for equality between `error_category` objects.
operator!=	Tests for inequality between `error_category` objects.
operator<	Tests if the error_category object is less than the `error_category` object passed in for comparison.

default_error_condition

Stores the error code value for an error condition object.

```
C++
```

```
virtual error_condition default_error_condition(int _Errval) const;
```

Parameters

`_Errval`

The error code value to store in the error_condition.

Return Value

Returns `error_condition(_Errval, *this)`.

Remarks

equivalent

Returns a value that specifies whether error objects are equivalent.

```
C++
```

```
virtual bool equivalent(value_type _Errval,
    const error_condition& _Cond) const;

virtual bool equivalent(const error_code& _Code,
    value_type _Errval) const;
```

Parameters

_Errval
The error code value to compare.

_Cond
The error_condition object to compare.

_Code
The error_code object to compare.

Return Value

`true` if the category and value are equal; otherwise, `false`.

Remarks

The first member function returns `*this == _Cond.category() && _Cond.value() == _Errval`.

The second member function returns `*this == _Code.category() && _Code.value() == _Errval`.

generic_category

```
C++
```

```
const error_category& generic_category();
```

message

Returns the name of the specified error code.

```
C++
```

```
virtual string message(error_code::value_type val) const = 0;
```

Parameters

val
The error code value to describe.

Return Value

Returns a descriptive name of the error code *val* for the category. If the error code is unrecognized, returns `"unknown error"`.

Remarks

name

Returns the name of the category.

```
C++

virtual const char *name() const = 0;
```

Return Value

Returns the name of the category as a null-terminated byte string.

operator=

```
C++

error_category& operator=(const error_category&) = delete;
```

operator==

Tests for equality between `error_category` objects.

```
C++

bool operator==(const error_category& right) const;
```

Parameters

right
The object to be tested for equality.

Return Value

`true` if the objects are equal; `false` if the objects aren't equal.

Remarks

This member operator returns `this == &right`.

operator!=

Tests for inequality between `error_category` objects.

```
C++
```

```
bool operator!=(const error_category& right) const;
```

Parameters

right
The object to be tested for inequality.

Return Value

`true` if the `error_category` object isn't equal to the `error_category` object passed in *right*; otherwise `false`.

Remarks

The member operator returns `(!*this == right)`.

`operator<`

Tests if the error_category object is less than the `error_category` object passed in for comparison.

```
C++
```

```
bool operator<(const error_category& right) const;
```

Parameters

right
The `error_category` object to be compared.

Return Value

`true` if the `error_category` object is less than the `error_category` object passed in for comparison; Otherwise, **`false`**.

Remarks

The member operator returns `this < &right`.

system_category

```
C++
```

```
const error_category& system_category();
```

value_type

A type that represents the stored error code value.

```
C++
```

```
typedef int value_type;
```

Remarks

This type definition is a synonym for **`int`**.

error_code Class

Article • 12/07/2021

Represents low-level system errors that are implementation-specific.

Syntax

```
C++
```

```
class error_code;
```

Remarks

An object of type `error_code` class stores an error code value and a pointer to an object that represents a category of error codes that describe reported low-level system errors.

Members

Constructors

Name	Description
error_code	Constructs an object of type `error_code`.

Typedefs

Name	Description
value_type	A type that represents the stored error code value.

Functions

Name	Description
assign	Assigns an error code value and category to an error code.
category	Returns the error category.

Name	Description
clear	Clears the error code value and category.
default_error_condition	Returns the default error condition.
message	Returns the name of the error code.

Operators

Name	Description
operator==	Tests for equality between `error_code` objects.
operator!=	Tests for inequality between `error_code` objects.
operator<	Tests if the `error_code` object is less than the `error_code` object passed in for comparison.
operator=	Assigns a new enumeration value to the `error_code` object.
operator bool	Casts a variable of type `error_code`.

assign

Assigns an error code value and category to an error code.

```
C++
```

```
void assign(value_type val, const error_category& _Cat);
```

Parameters

val
The error code value to store in the `error_code`.

_Cat
The error category to store in the `error_code`.

Remarks

The member function stores *val* as the error code value and a pointer to *_Cat*.

category

Returns the error category.

```
C++
```

```
const error_category& category() const;
```

Remarks

clear

Clears the error code value and category.

```
C++
```

```
clear();
```

Remarks

The member function stores a zero error code value and a pointer to the generic_category object.

default_error_condition

Returns the default error condition.

```
C++
```

```
error_condition default_error_condition() const;
```

Return Value

The error_condition specified by default_error_condition.

Remarks

This member function returns `category().default_error_condition(value())`.

error_code

Constructs an object of type `error_code`.

```
C++
```

```
error_code();

error_code(value_type val, const error_category& _Cat);

template <class _Enum>
error_code(_Enum _Errcode,
    typename enable_if<is_error_code_enum<_Enum>::value,
    error_code>::type* = 0);
```

Parameters

val
The error code value to store in the `error_code`.

_Cat
The error category to store in the `error_code`.

_Errcode
The enumeration value to store in the `error_code`.

Remarks

The first constructor stores a zero error code value and a pointer to the generic_category.

The second constructor stores *val* as the error code value and a pointer to error_category.

The third constructor stores `(value_type)_Errcode` as the error code value and a pointer to the generic_category.

message

Returns the name of the error code.

```
C++
```

```
string message() const;
```

Return Value

A `string` representing the name of the error code.

Remarks

This member function returns `category().message(value())`.

operator==

Tests for equality between `error_code` objects.

```
C++
```

```
bool operator==(const error_code& right) const;
```

Parameters

right
The object to be tested for equality.

Return Value

`true` if the objects are equal; **`false`** if objects are not equal.

Remarks

The member operator returns `category() == right.category() && value == right.value()`.

operator!=

Tests for inequality between `error_code` objects.

```
C++
```

```
bool operator!=(const error_code& right) const;
```

Parameters

right
The object to be tested for inequality.

Return Value

`true` if the `error_code` object is not equal to the `error_code` object passed in *right*; otherwise `false`.

Remarks

The member operator returns `!(*this == right)`.

`operator<`

Tests if the `error_code` object is less than the `error_code` object passed in for comparison.

```
C++

bool operator<(const error_code& right) const;
```

Parameters

right
The error_code object to be compared.

Return Value

`true` if the `error_code` object is less than the `error_code` object passed in for comparison; Otherwise, `false`.

Remarks

The member operator returns `category() < right.category() || category() == right.category() && value < right.value()`.

operator=

Assigns a new enumeration value to the `error_code` object.

```
C++
```

```
template <class _Enum>
typename enable_if<is_error_code_enum<_Enum>::value, error_code>::type&
    operator=(_Enum _Errcode);
```

Parameters

_Errcode
The enumeration value to assign to the `error_code` object.

Return Value

A reference to the `error_code` object that is being assigned the new enumeration value by the member function.

Remarks

The member operator stores `(value_type)_Errcode` as the error code value and a pointer to the generic_category. It returns `*this`.

operator bool

Casts a variable of type `error_code`.

```
C++
```

```
explicit operator bool() const;
```

Return Value

The Boolean value of the `error_code` object.

Remarks

The operator returns a value convertible to `true` only if value is not equal to zero. The return type is convertible only to `bool`, not to `void *` or other known scalar types.

value

Returns the stored error code value.

```
C++
```

```
value_type value() const;
```

Return Value

The stored error code value of type value_type.

value_type

A type that represents the stored error code value.

```
C++
```

```
typedef int value_type;
```

Remarks

This type definition is a synonym for `int`.

error_condition Class

Article • 12/07/2021

Represents user-defined error codes.

Syntax

```
C++
```

```
class error_condition;
```

Remarks

An object of type `error_condition` stores an error code value and a pointer to an object that represents a category of error codes used for reported user-defined errors.

Members

Constructors

Name	Description
error_condition	Constructs an object of type `error_condition`.

Typedefs

Name	Description
value_type	A type that represents the stored error code value.

Functions

Name	Description
assign	Assigns an error code value and category to an error condition.
category	Returns the error category.

Name	Description
clear	Clears the error code value and category.
message	Returns the name of the error code.

Operators

Name	Description
operator==	Tests for equality between `error_condition` objects.
operator!=	Tests for inequality between `error_condition` objects.
operator<	Tests if the `error_condition` object is less than the `error_code` object passed in for comparison.
operator=	Assigns a new enumeration value to the `error_condition` object.
operator bool	Casts a variable of type `error_condition`.

assign

Assigns an error code value and category to an error condition.

```
C++
```

```
void assign(value_type val, const error_category& _Cat);
```

Parameters

val
The error code value to store in the `error_code`.

_Cat
The error category to store in the `error_code`.

Remarks

The member function stores *val* as the error code value and a pointer to *_Cat*.

category

Returns the error category.

```
C++
```

```
const error_category& category() const;
```

Return Value

A reference to the stored error category

Remarks

clear

Clears the error code value and category.

```
C++
```

```
clear();
```

Remarks

The member function stores a zero error code value and a pointer to the generic_category object.

error_condition

Constructs an object of type `error_condition`.

```
C++
```

```
error_condition();

error_condition(value_type val, const error_category& _Cat);

template <class _Enum>
error_condition(_Enum _Errcode,
    typename enable_if<is_error_condition_enum<_Enum>::value,
    error_code>::type* = 0);
```

Parameters

val
The error code value to store in the `error_condition`.

_Cat
The error category to store in the `error_condition`.

_Errcode
The enumeration value to store in the `error_condition`.

Remarks

The first constructor stores a zero error code value and a pointer to the generic_category.

The second constructor stores *val* as the error code value and a pointer to error_category.

The third constructor stores `(value_type)_Errcode` as the error code value and a pointer to the generic_category.

message

Returns the name of the error code.

```
C++
```

```
string message() const;
```

Return Value

A `string` representing the name of the error code.

Remarks

This member function returns `category().message(value())`.

operator==

Tests for equality between `error_condition` objects.

```
C++
```

```
bool operator==(const error_condition& right) const;
```

Parameters

right
The ojbect to be tested for equality.

Return Value

`true` if the objects are equal; `false` if objects are not equal.

Remarks

The member operator returns `category() == right.category() && value == right.value()`.

operator!=

Tests for inequality between `error_condition` objects.

```
C++
bool operator!=(const error_condition& right) const;
```

Parameters

right
The object to be tested for inequality.

Return Value

`true` if the `error_condition` object is not equal to the `error_condition` object passed in *right*; otherwise `false`.

Remarks

The member operator returns `!(*this == right)`.

operator<

Tests if the `error_condition` object is less than the `error_code` object passed in for comparison.

```
C++
```

```
bool operator<(const error_condition& right) const;
```

Parameters

right
The `error_condition` object to be compared.

Return Value

`true` if the `error_condition` object is less than the `error_condition` object passed in for comparison; Otherwise, **`false`**.

Remarks

The member operator returns `category() < right.category() || category() == right.category() && value < right.value()`.

operator=

Assigns a new enumeration value to the `error_condition` object.

```
C++
```

```
template <class _Enum>
error_condition(_Enum error,
    typename enable_if<is_error_condition_enum<_Enum>::value,
    error_condition>::type&
    operator=(Enum _Errcode);
```

Parameters

_Errcode
The enumeration value to assign to the `error_condition` object.

Return Value

A reference to the `error_condition` object that is being assigned the new enumeration value by the member function.

Remarks

The member operator stores `(value_type)error` as the error code value and a pointer to the generic_category. It returns `*this`.

operator bool

Casts a variable of type `error_condition`.

```
C++

explicit operator bool() const;
```

Return Value

The Boolean value of the `error_condition` object.

Remarks

The operator returns a value convertible to `true` only if value is not equal to zero. The return type is convertible only to `bool`, not to `void *` or other known scalar types.

value

Returns the stored error code value.

```
C++

value_type value() const;
```

Return Value

The stored error code value of type value_type.

Remarks

value_type

A type that represents the stored error code value.

```
C++
```

```
typedef int value_type;
```

Remarks

The type definition is a synonym for `int`.

is_error_code_enum Class

Article • 08/03/2021

Represents a type predicate that tests for the error_code enumeration.

Syntax

```
C++
```

```
template <_Enum>
    class is_error_code_enum;
```

Remarks

An instance of this type predicate holds true if the type `_Enum` is an enumeration value suitable for storing in an object of type `error_code`.

It is permissible to add specializations to this type for user-defined types.

See also

<type_traits>

is_error_condition_enum Class

Article • 08/03/2021

Represents a type predicate that tests for the error_condition enumeration.

Syntax

```
C++
```

```
template <_Enum>
    class is_error_condition_enum;
```

Remarks

An instance of this type predicate holds true if the type `_Enum` is an enumeration value suitable for storing in an object of type `error_condition`.

It is permissible to add specializations to this type for user-defined types.

See also

<type_traits>

system_error Class

Article • 08/03/2021

Represents the base class for all exceptions thrown to report a low-level system error.

Syntax

C++

```
class system_error : public runtime_error {
    explicit system_error(error_code _Errcode, const string& _Message = "");
    system_error(error_code _Errcode, const char *_Message);
    system_error(error_code::value_type _Errval, const error_category&
_Errcat, const string& _Message);
    system_error(error_code::value_type _Errval, const error_category&
_Errcat, const char *_Message);

    const error_code& code() const throw();
    const char* what() const noexcept override;
};
```

Remarks

The value returned by `what` in the class exception is constructed from `_Message` and the stored object of type error_code (either `code` or `error_code(_Errval, _Errcat)`).

The member function `code` returns the stored error_code object.

<thread>

Article • 12/07/2021

Include the standard header <thread> to define the class `thread` and various supporting functions.

Syntax

```
C++

#include <thread>
```

Remarks

> **Note**
>
> In code that is compiled by using **/clr**, this header is blocked.

The `__STDCPP_THREADS__` macro is defined as a nonzero value to indicate that threads are supported by this header.

Members

Public Classes

Name	Description
thread Class	Defines an object that is used to observe and manage a thread of execution in an application.

Public Structures

Name	Description

Name	Description
hash Structure (C++ Standard Library)	Defines a member function that returns a value that is uniquely determined by a `thread::id`. The member function defines a hash function that is suitable for mapping values of type `thread::id` to a distribution of index values.

Public Functions

Name	Description
get_id	Uniquely identifies the current thread of execution.
sleep_for	Blocks the calling thread.
sleep_until	Blocks the calling thread at least until the specified time.
swap	Exchanges the states of two `thread` objects.
yield	Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

Public Operators

Name	Description
operator>= Operator	Determines whether one `thread::id` object is greater than or equal to another.
operator> Operator	Determines whether one `thread::id` object is greater than another.
operator<= Operator	Determines whether one `thread::id` object is less than or equal to another.
operator< Operator	Determines whether one `thread::id` object is less than another.
operator!= Operator	Compares two `thread::id` objects for inequality.
operator== Operator	Compares two `thread::id` objects for equality.
operator<< Operator	Inserts a text representation of a `thread::id` object into a stream.

See also

Header Files Reference

Thread Safety in the C++ Standard Library

`<thread>` functions

Article • 03/29/2022

get_id

Uniquely identifies the current thread of execution.

```
C++

thread::id this_thread::get_id() noexcept;
```

Return Value

An object of type thread::id that uniquely identifies the current thread of execution.

sleep_for

Blocks the calling thread.

```
C++

template <class Rep,
class Period>
inline void sleep_for(const chrono::duration<Rep, Period>& Rel_time);
```

Parameters

Rel_time
A duration object that specifies a time interval.

Remarks

The function blocks the calling thread for at least the time that's specified by *Rel_time*. This function does not throw any exceptions.

sleep_until

Blocks the calling thread at least until the specified time.

```
C++
```

```
template <class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& Abs_time);

void sleep_until(const xtime *Abs_time);
```

Parameters

Abs_time
Represents a point in time.

Remarks

This function does not throw any exceptions.

swap

Swaps the states of two `thread` objects.

```
C++
```

```
void swap(thread& Left, thread& Right) noexcept;
```

Parameters

Left
The left `thread` object.

Right
The right `thread` object.

Remarks

The function calls `Left.swap(Right)`.

yield

Signals the operating system to run other threads, even if the current thread would ordinarily continue to run.

C++

```
inline void yield() noexcept;
```

See also

<thread>

`<thread>` operators

Article • 12/07/2021

operator!=
operator>
operator>=
operator<
operator<<
operator<=
operator==

`operator>=`

Determines whether one `thread::id` object is greater than or equal to another.

```
C++
```

```
bool operator>= (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`!(Left < Right)`

Remarks

This function does not throw any exceptions.

`operator>`

Determines whether one `thread::id` object is greater than another.

```
C++
```

```
bool operator> (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`Right < Left`

Remarks

This function does not throw any exceptions.

`operator<=`

Determines whether one `thread::id` object is less than or equal to another.

```
C++
```

```
bool operator<= (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`!(Right < Left)`

Remarks

This function does not throw any exceptions.

`operator<`

Determines whether one `thread::id` object is less than another.

```
C++
```

```
bool operator<(
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`true` if *Left* precedes *Right* in the total ordering; otherwise, **`false`**.

Remarks

The operator defines a total ordering on all `thread::id` objects. These objects can be used as keys in associative containers.

This function does not throw any exceptions.

operator!=

Compares two `thread::id` objects for inequality.

```
C++
```

```
bool operator!= (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`!(Left == Right)`

Remarks

This function does not throw any exceptions.

operator==

Compares two `thread::id` objects for equality.

```
C++
```

```
bool operator== (
    thread::id Left,
    thread::id Right) noexcept
```

Parameters

Left
The left `thread::id` object.

Right
The right `thread::id` object.

Return Value

`true` if the two objects represent the same thread of execution or if neither object represents a thread of execution; otherwise, `false`.

Remarks

This function does not throw any exceptions.

`operator<<`

Inserts a text representation of a `thread::id` object into a stream.

```
C++
```

```
template <class Elem, class Tr>
basic_ostream<Elem, Tr>& operator<<(
    basic_ostream<Elem, Tr>& Ostr, thread::id Id);
```

Parameters

Ostr
A basic_ostream object.

Id
A `thread::id` object.

Return Value

Ostr.

Remarks

This function inserts *Id* into *Ostr*.

If two `thread::id` objects compare equal, the inserted text representations of those objects are the same.

See also

<thread>

hash Structure (C++ Standard Library)

Article • 02/07/2023

Defines a member function that returns a value that's uniquely determined by `Val`. The member function defines a hash function that's suitable for mapping values of type `thread::id` to a distribution of index values.

Syntax

C++

```
template <>
struct hash<thread::id> :
    public unary_function<thread::id, size_t>
{
    size_t operator()(thread::id Val) const;

};
```

Requirements

Header: <thread>

Namespace: std

See also

Header Files Reference
<thread>
unary_function Struct

`thread` Class

Article • 09/13/2024

Defines an object that's used to observe and manage a thread of execution within an application.

Syntax

```
C++
```

```
class thread;
```

Remarks

You can use a `thread` object to observe and manage a thread of execution within an application. A `thread` object that's created by using the default constructor isn't associated with any thread of execution. A `thread` object that's constructed by using a callable object creates a new thread of execution and calls the callable object in that `thread`. `Thread` objects can be moved but not copied, which is why a thread of execution can be associated with only one `thread` object.

Every thread of execution has a unique identifier of type `thread::id`. The function `this_thread::get_id` returns the identifier of the calling thread. The member function `thread::get_id` returns the identifier of the thread that's managed by a `thread` object. For a default-constructed `thread` object, the `thread::get_id` method returns an object that has a value that's the same for all default-constructed `thread` objects and different from the value that's returned by `this_thread::get_id` for any thread of execution that could be joined at the time of the call.

Members

Public Classes

Expand table

Name	Description
id	Uniquely identifies the associated thread.

Public Constructors

Expand table

Name	Description
thread	Constructs a `thread` object.

Public Methods

Expand table

Name	Description
detach	Detaches the associated thread from the `thread` object.
get_id	Returns the unique identifier of the associated thread.
hardware_concurrency	Static. Returns an estimate of the number of hardware thread contexts.
join	Blocks until the associated thread completes.
joinable	Specifies whether the associated thread is joinable.
native_handle	Returns the implementation-specific type that represents the thread handle.
swap	Swaps the object state with a specified `thread` object.

Public Operators

Expand table

Name	Description
thread::operator=	Associates a thread with the current `thread` object.

Requirements

Header: `<thread>`

Namespace: `std`

`detach`

Detaches the associated thread. The operating system becomes responsible for releasing thread resources on termination.

```
C++
```

```
void detach();
```

Remarks

After a call to `detach`, subsequent calls to get_id return id.

If the thread associated with the calling object isn't joinable, the function throws a system_error that has an error code of `invalid_argument`.

If the thread associated with the calling object is invalid, the function throws a `system_error` that has an error code of `no_such_process`.

`get_id`

Returns a unique identifier for the associated thread.

```
C++
```

```
id get_id() const noexcept;
```

Return value

An id object that uniquely identifies the associated thread, or `id()` if no thread is associated with the object.

`hardware_concurrency`

Static method that returns an estimate of the number of hardware thread contexts.

```
C++
```

```
static unsigned int hardware_concurrency() noexcept;
```

Return value

An estimate of the number of hardware thread contexts. If the value can't be computed or isn't well defined, this method returns 0.

Microsoft specific

`hardware_concurrency` returns the number of logical processors, which corresponds to the number of hardware threads that can execute simultaneously. It takes into account the number of physical processors, the number of cores in each physical processor, and simultaneous multithreading on each single core.

Before Windows 11 and Windows Server 2022, applications were limited by default to a single processor group, having at most 64 logical processors. This limited the number of concurrently executing threads to 64. For more information, see Processor Groups.

Starting with Windows 11 and Windows Server 2022, processes and their threads have processor affinities that by default span all processors in the system and across multiple groups on machines with more than 64 processors. The limit on the number of concurrent threads is now the total number of logical processors in the system.

`id` class

Provides a unique identifier for each thread of execution in the process.

C++

```
class thread::id {
    id() noexcept;
};
```

Remarks

The default constructor creates an object that doesn't compare equal to the `thread::id` object for any existing thread.

All default-constructed `thread::id` objects compare equal.

`join`

Blocks until the thread of execution associated with the calling object completes.

```
C++
```

```
void join();
```

Remarks

If the call succeeds, subsequent calls to get_id for the calling object return a default thread::id that doesn't compare equal to the `thread::id` of any existing thread; if the call doesn't succeed, the value that's returned by `get_id` is unchanged.

`joinable`

Specifies whether the associated thread is joinable.

```
C++
```

```
bool joinable() const noexcept;
```

Return value

`true` if the associated thread is joinable; otherwise, `false`.

Remarks

A thread object is joinable if `get_id() != id()`.

`native_handle`

Returns the implementation-specific type that represents the thread handle. The thread handle can be used in implementation-specific ways.

```
C++
```

```
native_handle_type native_handle();
```

Return value

`native_handle_type` is defined as a Win32 `HANDLE` cast as `void *`.

`thread::operator=`

Associates the thread of a specified object with the current object.

```
C++
```

```
thread& operator=(thread&& Other) noexcept;
```

Parameters

`Other`
A `thread` object.

Return value

`*this`

Remarks

The method calls detach if the calling object is joinable.

After the association is made, `Other` is set to a default-constructed state.

`swap`

Swaps the object state with that of a specified `thread` object.

```
C++
```

```
void swap(thread& Other) noexcept;
```

Parameters

`Other`
A `thread` object.

`thread` constructor

Constructs a `thread` object.

```
C++
```

```
thread() noexcept;
template <class Fn, class... Args>
explicit thread(Fn&& F, Args&&... A);

thread(thread&& Other) noexcept;
```

Parameters

`F`

An application-defined function to execute on the thread.

`A`

A list of arguments to be passed to *`F`*.

`Other`

An existing `thread` object.

Remarks

The first constructor constructs an object that's not associated with a thread of execution. The value returned by `get_id` for the constructed object is `thread::id()`.

The second constructor constructs an object that's associated with a new thread of execution. It executes the pseudo-function `INVOKE` defined in <functional>. If not enough resources are available to start a new thread, the function throws a system_error object that has an error code of `resource_unavailable_try_again`. If the call to *`F`* terminates with an uncaught exception, terminate is called.

The third constructor constructs an object that's associated with the thread that's associated with `Other`. `Other` is then set to a default-constructed state.

See also

Header Files Reference
<thread>

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`<tuple>`

Article • 12/07/2021

Defines a template `tuple` whose instances hold objects of varying types.

Requirements

Header: <tuple>

Namespace: std

Members

Classes and Structs

Name	Description
tuple Class	Wraps a fixed-length sequence of elements.
tuple_element Class	Wraps the type of a `tuple` element.
tuple_size Class	Wraps `tuple` element count.
uses_allocator	

Objects

Name	Description
tuple_element_t	
tuple_size_v	

Operators

Name	Description
operator==	Comparison of `tuple` objects, equal.
operator!=	Comparison of `tuple` objects, not equal.

Name	Description
operator<	Comparison of `tuple` objects, less than.
operator<=	Comparison of `tuple` objects, less than or equal.
operator>	Comparison of `tuple` objects, greater than.
operator>=	Comparison of `tuple` objects, greater than or equal.

Functions

Name	Description
apply	Calls a function with a tuple.
forward_as_tuple	Constructs a tuple of references.
get	Gets an element from a `tuple` object.
make_from_tuple	Shorthand to make a `tuple`.
make_tuple	Makes a `tuple` from element values.
swap	
tie	Makes a `tuple` from element references.
tuple_cat	Constructs a tuple object with a range of the type elements.

See also

<array>

`<tuple>` functions

Article • 12/07/2021

apply

```
C++
```

```
template <class F, class Tuple> constexpr decltype(auto) apply(F&& f,
Tuple&& t);
```

Remarks

Calls function *F* with a tuple *t*.

forward_as_tuple

```
C++
```

```
template <class... TTypes>
    constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&...) noexcept;
```

Return Value

Returns `tuple<TTypes&&...>(std::forward<TTypes>(t)...)`.

Remarks

Constructs a tuple of references to the arguments in *t* suitable for forwarding as arguments to a function.

get

Gets an element from a `tuple` object, by index or (in C++14) by type.

```
C++
```

```
// by index:
// get reference to Index element of tuple
template <size_t Index, class... Types>
```

```
   constexpr tuple_element_t<Index, tuple<Types...>>& get(tuple<Types...>&
Tuple) noexcept;

// get const reference to Index element of tuple
template <size_t Index, class... Types>
   constexpr const tuple_element_t<Index, tuple<Types...>>& get(const
tuple<Types...>& Tuple) noexcept;

// get rvalue reference to Index element of tuple
template <size_t Index, class... Types>
   constexpr tuple_element_t<Index, tuple<Types...>>&& get(tuple<Types...>&&
Tuple) noexcept;

// (C++14) by type:
// get reference to T element of tuple
template <class T, class... Types>
   constexpr T& get(tuple<Types...>& Tuple) noexcept;

// get const reference to T element of tuple
template <class T, class... Types>
   constexpr const T& get(const tuple<Types...>& Tuple) noexcept;

// get rvalue reference to T element of tuple
template <class T, class... Types>
   constexpr T&& get(tuple<Types...>&& Tuple) noexcept;
```

Parameters

Index
The index of the element to get.

Types
The sequence of types declared in the tuple, in declaration order.

T
The type of the element to get.

Tuple
A `std::tuple` that contains any number of elements.

Remarks

The template functions return a reference to the value at index *Index*, or of type *T* in the `tuple` object.

Calling `get<T>(Tuple)` will produce a compiler error if Tuple contains more or less than one element of type T.

Example

C++

```
#include <tuple>
#include <iostream>
#include <string>

using namespace std;

int main() {
    tuple<int, double, string> tup(0, 1.42, "Call me Tuple");

    // get elements by index
    cout << " " << get<0>(tup);
    cout << " " << get<1>(tup);
    cout << " " << get<2>(tup) << endl;

    // get elements by type
    cout << " " << get<int>(tup);
    cout << " " << get<double>(tup);
    cout << " " << get<string>(tup) << endl;
}
```

Output

```
0 1.42 Call me Tuple
0 1.42 Call me Tuple
```

make_from_tuple

C++

```
template <class T, class Tuple> constexpr T make_from_tuple(Tuple&& t);
```

Remarks

Same as `return make_from_tuple_impl<T>(forward<Tuple>(t), make_index_sequence<tuple_size_v<decay_t<Tuple>>>{})`.

make_tuple

Makes a `tuple` from element values.

C++

```
template <class T1, class T2, ..., class TN>
   tuple<V1, V2, ..., VN> make_tuple(const T1& t1, const T2& t2, ..., const
TN& tN);
```

Parameters

TN
The type of the Nth function parameter.

tN
The value of the Nth function parameter.

Remarks

The template function returns `tuple<V1, V2, ..., VN>(t1, t2, ..., tN)`, where each type `Vi` is `X&` when the corresponding type `Ti` is `cv reference_wrapper<X>`; otherwise, it is `Ti`.

One advantage of `make_tuple` is that the types of objects that are being stored are determined automatically by the compiler and do not have to be explicitly specified. Don't use explicit template arguments such as `make_tuple<int, int>(1, 2)` when you use `make_tuple` because it is unnecessarily verbose and adds complex rvalue reference problems that might cause compilation failure.

Example

C++

```
// std__tuple__make_tuple.cpp
// compile by using: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0) << std::endl;
```

```
    c0 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0) << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
4 5 6 7
```

swap

C++

```
template <class... Types>
    void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below );
```

tie

Makes a `tuple` from element references.

C++

```
template <class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN&> tie(T1& t1, T2& t2, ..., TN& tN);
```

Parameters

TN

The base type of the Nth tuple element.

Remarks

The template function returns `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tN)`.

Example

C++

```
// std__tuple__tie.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    int v4 = 4;
    double v5 = 5;
    int v6 = 6;
    double v7 = 7;
    std::tie(v4, v5, v6, v7) = c0;

// display contents " 0 1 2 3"
    std::cout << " " << v4;
    std::cout << " " << v5;
    std::cout << " " << v6;
    std::cout << " " << v7;
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
```

tuple_cat

C++

```
template <class... Tuples> constexpr tuple<CTypes...>
tuple_cat(Tuples&&...);
```

Return Value

A tuple object constructed by initializing the each type element.

tuple_element_t

```
C++
```

```
template <size_t I, class T>
    using tuple_element_t = typename tuple_element<I, T>::type;
```

tuple_size_v

```
C++
```

```
template <class T>
    inline constexpr size_t tuple_size_v = tuple_size<T>::value;
```

`<tuple>` operators

Article • 12/07/2021

operator!=

Compare `tuple` objects for inequality.

```
C++
```

```
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
bool operator!=(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns false when `N` is 0, otherwise `get<0>(tpl1) != get<0>(tpl2) || get<1>(tpl1) != get<1>(tpl2) || ... || get<N - 1>(tpl1) == get<N - 1>(tpl2)`.

Example

```
C++
```

```
// std__tuple__operator_ne.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;
```

```
    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 != c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 != c1);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
false
true
```

`operator<`

Compare `tuple` objects for less.

C++

```
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
bool operator<(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns true when `N` is greater than 0 and the first differing value in `tpl1` compares less than the corresponding value in `tpl2`, otherwise it returns false.

Example

C++

```
// std__tuple__operator_lt.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 < c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 < c1);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
false
true
```

`operator<=`

Compare `tuple` objects for less or equal.

```
C++
```

```
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
bool operator<=(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN
The type of the Nth tuple element.

Remarks

The function returns `!(tpl2 < tpl1)`.

Example

```
C++
```

```
// std__tuple__operator_le.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
```

```
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 <= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 <= c0);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
true
false
```

operator==

Compare `tuple` objects for equality.

C++

```
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
bool operator==(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN
The type of the Nth tuple element.

Remarks

The function returns true when `N` is 0, otherwise `get<0>(tpl1) == get<0>(tpl2) && get<1>(tpl1) == get<1>(tpl2) && ... && get<N - 1>(tpl1) == get<N - 1>(tpl2)`.

Example

C++

```cpp
// std__tuple__operator_eq.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 == c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 == c1);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
true
false
```

operator>

Compare `tuple` objects for greater.

C++

```cpp
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
```

```
bool operator>(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN
The type of the Nth tuple element.

Remarks

The function returns `tpl2 < tpl1`.

Example

C++

```
// std__tuple__operator_gt.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 > c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c1 > c0);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
false
true
```

`operator>=`

Compare `tuple` objects for greater or equal.

C++

```
template <class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
bool operator>=(const tuple<T1, T2, ..., TN>& tpl1,
    const tuple<U1, U2, ..., UN>& tpl2);
```

Parameters

TN

The type of the Nth tuple element.

Remarks

The function returns `!(tpl1 < tpl2)`.

Example

C++

```
// std__tuple__operator_ge.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>

typedef std::tuple<int, double, int, double> Mytuple;
int main() {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
```

```
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1 = std::make_tuple(4, 5, 6, 7);

// display contents " 4 5 6 7"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

// display results of comparisons
    std::cout << std::boolalpha << " " << (c0 >= c0);
    std::cout << std::endl;
    std::cout << std::boolalpha << " " << (c0 >= c1);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
true
false
```

tuple Class

Article • 06/21/2022

Wraps a fixed-length sequence of elements.

Syntax

C++

```
class tuple {
   tuple();
   explicit tuple(P1, P2, ..., PN); // 0 < N
   tuple(const tuple&);
   template <class U1, class U2, ..., class UN>
      tuple(const tuple<U1, U2, ..., UN>&);
   template <class U1, class U2>
      tuple(const pair<U1, U2>&); // N == 2

   void swap(tuple& right);
   tuple& operator=(const tuple&);
   template <class U1, class U2, ..., class UN>
      tuple& operator=(const tuple<U1, U2, ..., UN>&);
   template <class U1, class U2>
      tuple& operator=(const pair<U1, U2>&); // N == 2
};
```

Parameters

TN
The type of the Nth tuple element.

Remarks

The class template describes an object that stores N objects of types `T1`, `T2`, ..., `TN`, respectively, where `0 <= N <= Nmax`. The extent of a tuple instance `tuple<T1, T2, ..., TN>` is the number `N` of its template arguments. The index of the template argument `Ti` and of the corresponding stored value of that type is `i - 1`. Thus, while we number the types from 1 to N in this documentation, the corresponding index values range from 0 to N - 1.

Example

C++

```cpp
// tuple.cpp
// compile with: /EHsc

#include <vector>
#include <iomanip>
#include <iostream>
#include <tuple>
#include <string>

using namespace std;

typedef tuple <int, double, string> ids;

void print_ids(const ids& i)
{
   cout << "( "
        << get<0>(i) << ", "
        << get<1>(i) << ", "
        << get<2>(i) << " )." << endl;
}

int main( )
{
   // Using the constructor to declare and initialize a tuple
   ids p1(10, 1.1e-2, "one");

   // Compare using the helper function to declare and initialize a tuple
   ids p2;
   p2 = make_tuple(10, 2.22e-1, "two");

   // Making a copy of a tuple
   ids p3(p1);

   cout.precision(3);
   cout << "The tuple p1 is: ( ";
   print_ids(p1);
   cout << "The tuple p2 is: ( ";
   print_ids(p2);
   cout << "The tuple p3 is: ( ";
   print_ids(p3);

   vector<ids> v;

   v.push_back(p1);
   v.push_back(p2);
   v.push_back(make_tuple(3, 3.3e-2, "three"));

   cout << "The tuples in the vector are" << endl;
   for(vector<ids>::const_iterator i = v.begin(); i != v.end(); ++i)
   {
      print_ids(*i);
```

```
    }
}
```

Output

```
The tuple p1 is: ( 10, 0.011, one ).
The tuple p2 is: ( 10, 0.222, two ).
The tuple p3 is: ( 10, 0.011, one ).
The tuples in the vector are
( 10, 0.011, one ).
( 10, 0.222, two ).
( 3, 0.033, three ).
```

operator=

Assigns a `tuple` object.

C++

```
tuple& operator=(const tuple& right);

template <class U1, class U2, ..., class UN>
   tuple& operator=(const tuple<U1, U2, ..., UN>& right);

template <class U1, class U2>
   tuple& operator=(const pair<U1, U2>& right); // N == 2

tuple& operator=(tuple&& right);

template <class U1, class U2>
   tuple& operator=(pair<U1, U2>&& right);
```

Parameters

UN
The type of the Nth copied tuple element.

right
The tuple to copy from.

Remarks

The first two member operators assign the elements of *right* to the corresponding elements of `*this`. The third member operator assigns `right.first` to the element at

index 0 of `*this` and `right.second` to the element at index 1. All three member operators return `*this`.

The remaining member operators are analogs to earlier ones, but with Rvalue Reference Declarator: &&.

Example

C++

```
// std__tuple__tuple_operator_as.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>
#include <utility>

typedef std::tuple<int, double, int, double> Mytuple;
int main()
    {
    Mytuple c0(0, 1, 2, 3);

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c0);
    std::cout << " " << std::get<1>(c0);
    std::cout << " " << std::get<2>(c0);
    std::cout << " " << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1;
    c1 = c0;

// display contents " 0 1 2 3"
    std::cout << " " << std::get<0>(c1);
    std::cout << " " << std::get<1>(c1);
    std::cout << " " << std::get<2>(c1);
    std::cout << " " << std::get<3>(c1);
    std::cout << std::endl;

    std::tuple<char, int> c2;
    c2 = std::make_pair('x', 4);

// display contents " x 4"
    std::cout << " " << std::get<0>(c2);
    std::cout << " " << std::get<1>(c2);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
x 4
```

swap

Exchanges the elements of two tuples.

C++

```
template <class... Types>
   void swap(tuple<Types...&> left, tuple<Types...&> right);
```

Parameters

left
A tuple whose elements are to be exchanged with those of the tuple *right*.

right
A tuple whose elements are to be exchanged with those of the tuple *left*.

Remarks

The function executes `left.swap(right)`.

tuple

Constructs a `tuple` object.

C++

```
constexpr tuple();
explicit constexpr tuple(const Types&...);
template <class... UTypes>
   explicit constexpr tuple(UTypes&&...);

tuple(const tuple&) = default;
tuple(tuple&&) = default;

template <class... UTypes>
   constexpr tuple(const tuple<UTypes...>&);
template <class... UTypes>
   constexpr tuple(tuple<UTypes...>&&);
```

```
// only if sizeof...(Types) == 2
template <class U1, class U2>
   constexpr tuple(const pair<U1, U2>&);
template <class U1, class U2>
   constexpr tuple(pair<U1, U2>&&);
```

Parameters

UN
The type of the Nth copied tuple element.

right
The tuple to copy from.

Remarks

The first constructor constructs an object whose elements are default constructed.

The second constructor constructs an object whose elements are copy constructed from the arguments `P1`, `P2`, ..., `PN` with each `Pi` initializing the element at index `i - 1`.

The third and fourth constructors construct an object whose elements are copy constructed from the corresponding element of *right*.

The fifth constructor constructs an object whose element at index 0 is copy constructed from `right.first` and whose element at index 1 is copy constructed from `right.second`.

The remaining constructors are analogs to earlier ones, but with Rvalue Reference Declarator: &&.

Example

C++

```
// std__tuple__tuple_tuple.cpp
// compile with: /EHsc
#include <tuple>
#include <iostream>
#include <utility>

typedef std::tuple<int, double, int, double> Mytuple;
int main()
{
    Mytuple c0(0, 1, 2, 3);

    // display contents "0 1 2 3"
```

```
    std::cout << std::get<0>(c0) << " ";
    std::cout << std::get<1>(c0) << " ";
    std::cout << std::get<2>(c0) << " ";
    std::cout << std::get<3>(c0);
    std::cout << std::endl;

    Mytuple c1;
    c1 = c0;

    // display contents "0 1 2 3"
    std::cout << std::get<0>(c1) << " ";
    std::cout << std::get<1>(c1) << " ";
    std::cout << std::get<2>(c1) << " ";
    std::cout << std::get<3>(c1);
    std::cout << std::endl;

    std::tuple<char, int> c2(std::make_pair('x', 4));

    // display contents "x 4"
    std::cout << std::get<0>(c2) << " ";
    std::cout << std::get<1>(c2);
    std::cout << std::endl;

    Mytuple c3(c0);

    // display contents "0 1 2 3"
    std::cout << std::get<0>(c3) << " ";
    std::cout << std::get<1>(c3) << " ";
    std::cout << std::get<2>(c3) << " ";
    std::cout << std::get<3>(c3);
    std::cout << std::endl;

    typedef std::tuple<int, float, int, float> Mytuple2;
    Mytuple c4(Mytuple2(4, 5, 6, 7));

    // display contents "4 5 6 7"
    std::cout << std::get<0>(c4) << " ";
    std::cout << std::get<1>(c4) << " ";
    std::cout << std::get<2>(c4) << " ";
    std::cout << std::get<3>(c4);
    std::cout << std::endl;

    return (0);
}
```

Output

```
0 1 2 3
0 1 2 3
x 4
0 1 2 3
4 5 6 7
```

tuple_element Class

Article • 12/07/2021

Wraps a `tuple` element. Specializations wrap `array` elements and `pair` elements.

Syntax

C++

```
// CLASS tuple_element (find element by index)
template <size_t Index, class Tuple>
   struct tuple_element;

// tuple_element for const
template <size_t Index, class Tuple>
   struct tuple_element<Index, const Tuple>;

// tuple_element for volatile
template <size_t Index, class Tuple>
   struct tuple_element<Index, volatile Tuple>;

// tuple_element for const volatile
template <size_t Index, class Tuple>
   struct tuple_element<Index, const volatile Tuple>;

// Helper typedef
template <size_t Index, class Tuple>
   using tuple_element_t = typename tuple_element<Index, Tuple>::type;

// Specialization for arrays
template <size_t Index, class Elem, size_t Size>
   struct tuple_element<Index, array<Elem, Size>>;

// Specializations for pairs
// struct to determine type of element 0 in pair
template <class T1, class T2>
   struct tuple_element<0, pair<T1, T2>>;

// struct to determine type of element 1 in pair
template <class T1, class T2>
   struct tuple_element<1, pair<T1, T2>>;
```

Parameters

Index

The index of the designated element.

Tuple
The type of the tuple.

Elem
The type of an array element.

Size
The size of the array.

T1
The type of the first element in a pair.

T2
The type of the second element in a pair.

Remarks

The class template `tuple_element` has a nested typedef `type` that is a synonym for the type at index *Index* of the tuple type *Tuple*.

The typedef `tuple_element_t` is a convenient alias for `tuple_element<Index, Tuple>::type`.

The class template specialization for arrays provides an interface to an `array` as a tuple of `Size` elements, each of which has the same type. Each specialization has a nested typedef `type` that is a synonym for the type of the *Index* element of the `array`, with any const-volatile qualifications preserved.

The template specializations for `pair` types each provide a single member typedef, `type`, which is a synonym for the type of the element at the specified position in the pair, with any const and/or volatile qualifications preserved. The typedef `tuple_element_t` is a convenient alias for `tuple_element<N, pair<T1, T2>>::type`.

Use the get function to return the element at a specified position, or of a specified type.

Example: Get an element from a tuple

C++

```
#include <tuple>
#include <string>
#include <iostream>
```

```
using namespace std;
typedef tuple<int, double, string> MyTuple;

int main() {
    MyTuple c0{ 0, 1.5, "Tail" };

    tuple_element_t<0, MyTuple> val = get<0>(c0); //get by index
    tuple_element_t<1, MyTuple> val2 = get<1>(c0);
    tuple_element_t<2, MyTuple> val3 = get<string>(c0); // get by type

    cout << val << " " << val2 << " " << val3 << endl;
}
```

Output

```
0 1.5 Tail
```

Example: Get an element from an array

C++

```
#include <array>
#include <iostream>

using namespace std;
typedef array<int, 4> MyArray;

int main()
{
    MyArray c0 { 0, 1, 2, 3 };

    for (const auto& e : c0)
    {
        cout << e << " ";
    }
    cout << endl;

    // display first element "0"
    tuple_element<0, MyArray>::type val = c0.front();
    cout << val << endl;
}
```

Output

```
0 1 2 3
0
```

Example: Get an element from a pair

C++

```
#include <utility>
#include <iostream>

using namespace std;

typedef pair<int, double> MyPair;
int main() {
    MyPair c0(0, 1.333);

    // display contents "0 1"
    cout << get<0>(c0) << " ";
    cout << get<1>(c0) << endl;

    // display first element "0 " by index
    tuple_element<0, MyPair>::type val = get<0>(c0);
    cout << val << " ";

    // display second element by type
    tuple_element<1, MyPair>::type val2 = get<double>(c0);
    cout << val2 << endl;
}
```

Output

```
0 1.333
0 1.333
```

Requirements

Header: <tuple>

Header: <array> (for array specialization)

Header: <utility> (for pair specializations)

Namespace: std

`tuple_size` class

Article • 06/30/2022

Reports the number of elements that a `tuple` contains.

Syntax

C++

```
// TEMPLATE STRUCT tuple_size
template <class Tuple>
   struct tuple_size;

// number of elements in array
template <class Elem, size_t Size>
   struct tuple_size<array<Elem, Size>>
      : integral_constant<size_t, Size>;

// size of pair
template <class T1, class T2>
   struct tuple_size<pair<T1, T2>>
      : integral_constant<size_t, 2>

// size of tuple
template <class... Types>
   struct tuple_size<tuple<Types...>>
      : integral_constant<size_t, sizeof...(Types)>;

// size of const tuple
template <class Tuple>
   struct tuple_size<const Tuple>;

// size of volatile tuple
template <class Tuple>
   struct tuple_size<volatile Tuple>;

// size of const volatile tuple
template <class Tuple>
   struct tuple_size<const volatile Tuple>;

template <class T> inline constexpr size_t tuple_size_v =
tuple_size<T>::value;
```

Parameters

`Tuple`

The type of the tuple.

Elem

The type of the array elements.

Size

The size of the array.

T1

The type of the first member of the pair.

T2

The type of the second member of the pair.

Types

The types of the tuple elements.

Remarks

The class template has a member `value` that is an integral constant expression whose value is the extent of the tuple type *Tuple*.

The template specialization for arrays has a member `value` that is an integral constant expression whose value is *Size*, which is the size of the array.

The template specialization for pair has a member `value` that is an integral constant expression whose value is 2.

Example

C++

```
#include <tuple>
#include <iostream>

using namespace std;

typedef tuple<int, double, int, double> MyTuple;
int main()
{
    MyTuple c0(0, 1.5, 2, 3.7);

    // display contents "0 1 2 3"
    cout << get<0>(c0);
    cout << " " << get<1>(c0);
    cout << " " << get<2>(c0);
    cout << " " << get<3>(c0) << endl;
```

```
    // display size "4"
    cout << " " << tuple_size<MyTuple>::value << endl;
}
```

Output

```
0 1.5 2 3.7
4
```

Requirements

Header: `<tuple>`

Header: `<array>` (for array specialization)

Header: `<utility>` (for pair specialization)

Namespace: `std`

`<type_traits>`

Article • 12/07/2021

Defines templates for compile-time constants that give information about the properties of their type arguments, or produce transformed types.

Syntax

```
C++

#include <type_traits>
```

Remarks

The classes and templates in <type_traits> are used to support type inference, classification, and transformation at compile time. They are also used to detect type-related errors, and to help you optimize your generic code. Unary type traits describe a property of a type, binary type traits describe a relationship between types, and transformation traits modify a property of a type.

The helper class `integral_constant` and its template specializations `true_type` and `false_type` form the base classes for type predicates. A *type predicate* is a template that takes one or more type arguments. When a type predicate *holds true*, it's publicly derived, directly or indirectly, from true_type. When a type predicate *holds false*, it's publicly derived, directly or indirectly, from false_type.

A *type modifier* or *transformation trait* is a template that takes one or more template arguments and has one member, `type`, which is a synonym for the modified type.

Alias Templates

To simplify type traits expressions, alias templates for `typename some_trait<T>::type` are provided, where *some_trait* is the class template name. For example, add_const has an alias template for its type, `add_const_t`, defined as:

```
C++

template <class T>
using add_const_t = typename add_const<T>::type;
```

These are the provided aliases for the `type` members:

`add_const_t`
`add_cv_t`
`add_lvalue_reference_t`
`add_pointer_t`
`add_rvalue_reference_t`
`add_volatile_t`
`aligned_storage_t`
`aligned_union_t`\

`common_type_t`
`conditional_t`
`decay_t`
`enable_if_t`
`invoke_result_t`
`make_signed_t`
`make_unsigned_t`
`remove_all_extents_t`\

`remove_const_t`
`remove_cv_t`
`remove_extent_t`
`remove_pointer_t`
`remove_reference_t`
`remove_volatile_t`
`result_of_t`
`underlying_type_t`\

Classes

Helper class and typedefs

Name	Description
integral_constant	Makes an integral constant from a type and a value.
true_type	Holds integral constant with true value.
false_type	Holds integral constant with false value.

Primary type categories

Name	Description
is_void	Tests whether the type is `void`.
is_null_pointer	Tests whether the type is `std::nullptr_t`.
is_integral	Tests whether the type is integral.
is_floating_point	Tests whether the type is floating-point.
is_array	Tests whether the type is an array.
is_pointer	Tests whether the type is a pointer.
is_lvalue_reference	Tests if type is an lvalue reference.
is_rvalue_reference	Tests if type is an rvalue reference.
is_member_object_pointer	Tests whether the type is a pointer to a member object.
is_member_function_pointer	Tests whether the type is a pointer to a member function.
is_enum	Tests whether the type is an enumeration.
is_union	Tests whether the type is a union.
is_class	Tests whether the type is a class.
is_function	Tests whether the type is a function type.

Composite type categories

Name	Description
is_reference	Tests whether the type is a reference.
is_arithmetic	Tests whether the type is arithmetic.
is_fundamental	Tests whether the type is `void` or arithmetic.
is_object	Tests whether the type is an object type.
is_scalar	Tests whether the type is scalar.
is_compound	Tests whether the type is not scalar.
is_member_pointer	Tests whether the type is a pointer to a member.

Type properties

Name	Description
is_const	Tests whether the type is `const`.
is_volatile	Tests whether the type is `volatile`.
is_trivial	Tests whether the type is trivial.
is_trivially_copyable	Tests whether the type is trivially copyable.
is_standard_layout	Tests if type is a standard layout type.
is_pod	Tests whether the type is a POD.
is_literal_type	Tests whether the type can be a `constexpr` variable or used in a `constexpr` function.
is_empty	Tests whether the type is an empty class.
is_polymorphic	Tests whether the type is a polymorphic class.
is_abstract	Tests whether the type is an abstract class.
is_final	Tests whether the type is a class type marked `final`.
is_aggregate	
is_signed	Tests whether the type is a signed integer.
is_unsigned	Tests whether the type is an unsigned integer.
is_constructible	Tests whether the type is constructible using the specified argument types.
is_default_constructible	Tests whether the type has a default constructor.
is_copy_constructible	Tests whether the type has a copy constructor.
is_move_constructible	Tests whether the type has a move constructor.
is_assignable	Tests whether the first type can be assigned a value of the second type.
is_copy_assignable	Tests whether a type can be assigned a const reference value of the type.
is_move_assignable	Tests whether a type can be assigned an rvalue reference of the type.
is_swappable	
is_swappable_with	

Name	Description
is_destructible	Tests whether the type is destructible.
is_trivially_constructible	Tests whether the type uses no non-trivial operations when constructed using the specified types.
is_trivially_default_constructible	Tests whether the type uses no non-trivial operations when default constructed.
is_trivially_copy_constructible	Tests whether the type uses no non-trivial operations when copy constructed.
is_trivially_move_constructible	Tests whether the type uses no non-trivial operations when move constructed.
is_trivially_assignable	Tests whether the types are assignable and the assignment uses no non-trivial operations.
is_trivially_copy_assignable	Tests whether the type is copy assignable and the assignment uses no non-trivial operations.
is_trivially_move_assignable	Tests whether the type is move assignable and the assignment uses no non-trivial operations.
is_trivially_destructible	Tests whether the type is destructible and the destructor uses no non-trivial operations.
is_nothrow_constructible	Tests whether the type is constructible and is known not to throw when constructed using the specified types.
is_nothrow_default_constructible	Tests whether the type is default constructible and is known not to throw when default constructed.
is_nothrow_copy_constructible	Tests whether the type is copy constructible and the copy constructor is known not to throw.
is_nothrow_move_constructible	Tests whether the type is move constructible and the move constructor is known not to throw.
is_nothrow_assignable	Tests whether the type is assignable using the specified type and the assignment is known not to throw.
is_nothrow_copy_assignable	Tests whether the type is copy assignable and the assignment is known not to throw.
is_nothrow_move_assignable	Tests whether the type is move assignable and the assignment is known not to throw.
is_nothrow_swappable	
is_nothrow_swappable_with	

Name	Description
is_nothrow_destructible	Tests whether the type is destructible and the destructor is known not to throw.
`has_virtual_destructor`	Tests whether the type has a virtual destructor.
`has_unique_object_representations`	
is_invocable	Tests whether a callable type can be invoked using the specified argument types. Added in C++17.
is_invocable_r	Tests whether a callable type can be invoked using the specified argument types and the result is convertible to the specified type. Added in C++17.
is_nothrow_invocable	Tests whether a callable type can be invoked using the specified argument types and is known not to throw exceptions. Added in C++17.
is_nothrow_invocable_r	Tests whether a callable type can be invoked using the specified argument types and is known not to throw exceptions, and the result is convertible to the specified type. Added in C++17.

Type property queries

Name	Description
alignment_of	Gets the alignment of a type.
rank	Gets the number of array dimensions.
extent	Gets the number of elements in the specified array dimension.

Type relations

Name	Description
is_same	Tests whether two types are the same.
is_base_of	Tests whether one type is a base of another.
is_convertible	Tests whether one type is convertible to another.

Const-volatile modifications

Name	Description
add_const	Produces a `const` type from type.
add_volatile	Produces a `volatile` type from type.
add_cv	Produces a `const volatile` type from type.
remove_const	Produces a non-const type from type.
remove_volatile	Produces a non-volatile type from type.
remove_cv	Produces a non-const non-volatile type from type.

Reference modifications

Name	Description
add_lvalue_reference	Produces a reference to type from type.
add_rvalue_reference	Produces an rvalue reference to type from type
remove_reference	Produces a non-reference type from type.

Sign modifications

Name	Description
make_signed	Produces the type if signed, or the smallest signed type greater than or equal in size to type.
make_unsigned	Produces the type if unsigned, or the smallest unsigned type greater than or equal in size to type.

Array modifications

Name	Description
remove_all_extents	Produces a non-array type from an array type.
remove_extent	Produces the element type from an array type.

Pointer modifications

Name	Description
add_pointer	Produces a pointer to type from type.
remove_pointer	Produces a type from a pointer to type.

Other transformations

Name	Description
aligned_storage	Allocates uninitialized memory for an aligned type.
aligned_union	Allocates uninitialized memory for an aligned union with a non-trivial constructor or destructor.
common_type	Produces the common type of all the types of the parameter pack.
conditional	If the condition is true, produces the first specified type, otherwise the second specified type.
decay	Produces the type as passed by value. Makes non-reference, non-const, or non-volatile type, or makes a pointer to type.
enable_if	If the condition is true, produces the specified type, otherwise no type.
invoke_result	Determines the return type of the callable type that takes the specified argument types. Added in C++17.
result_of	Determines the return type of the callable type that takes the specified argument types. Added in C++14, deprecated in C++17.
underlying_type	Produces the underlying integral type for an enumeration type.

Logical operator traits

Name	Description
conjunction	
disjunction	
negation	

See also

<functional>

add_const Class

Article • 08/03/2021

Makes const type from type.

Syntax

C++

```
template <class Ty>
struct add_const;
```

Parameters

Ty
The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *Ty* if *Ty* is a reference, a function, or a const-qualified type, otherwise `const Ty`.

Example

C++

```
// std__type_traits__add_const.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
{
    std::add_const<int>::type *p = (const int *)0;

    p = p;  // to quiet "unused" warning
    std::cout << "add_const<int> == "
        << typeid(*p).name() << std::endl;

    return (0);
}
```

Output

```
add_const<int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_const Class

add_cv Class

Article • 08/03/2021

Makes `const volatile` type from type.

Syntax

C++

```
template <class T>
struct add_cv;

template <class T>
using add_cv_t = typename add_cv<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of the modified type `add_cv<T>` has a `type` member `typedef` equivalent to *T* modified by both add_volatile and add_const, unless *T* already has the cv-qualifiers, is a reference, or is a function.

The `add_cv_t<T>` helper type is a shortcut to access the `add_cv<T>` member typedef `type`.

Example

C++

```
// add_cv.cpp
// compile by using: cl /EHsc /W4 add_cv.cpp
#include <type_traits>
#include <iostream>

struct S {
    void f() {
        std::cout << "invoked non-cv-qualified S.f()" << std::endl;
    }
```

```
    void f() const {
        std::cout << "invoked const S.f()" << std::endl;
    }
    void f() volatile {
        std::cout << "invoked volatile S.f()" << std::endl;
    }
    void f() const volatile {
        std::cout << "invoked const volatile S.f()" << std::endl;
    }
};

template <class T>
void invoke() {
    T t;
    ((T *)&t)->f();
}

int main()
{
    invoke<S>();
    invoke<std::add_const<S>::type>();
    invoke<std::add_volatile<S>::type>();
    invoke<std::add_cv<S>::type>();
}
```

Output

```
invoked non-cv-qualified S.f()
invoked const S.f()
invoked volatile S.f()
invoked const volatile S.f()
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_const Class
remove_volatile Class

add_lvalue_reference Class

Article • 08/03/2021

Makes reference to type from type.

Syntax

C++

```
template <class T>
struct add_lvalue_reference;

template <class T>
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if *T* is an lvalue reference, otherwise `T&`.

Example

C++

```
#include <type_traits>
#include <iostream>

using namespace std;
int main()
{
    int val = 0;
    add_lvalue_reference_t<int> p = (int&)val;
    p = p;  // to quiet "unused" warning
    cout << "add_lvalue_reference_t<int> == "
        << typeid(p).name() << endl;

    return (0);
}
```

Output

```
add_lvalue_reference_t<int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_reference Class

add_rvalue_reference Class

Article • 08/03/2021

Creates an rvalue reference type of the template parameter, if it is an object or function type. Otherwise, because of the semantics of reference collapsing, the type is the same as the template parameter.

Syntax

C++

```
template <class T>
struct add_rvalue_reference;

template <class T>
using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;
```

Parameters

T
The type to modify.

Remarks

The `add_rvalue_reference` class has a member named `type`, which is an alias for the type of an rvalue reference to the template parameter *T*. The semantics of reference collapsing imply that, for non-object and non-function types *T*, `T&&` is a *T*. For example, when *T* is an lvalue reference type, `add_rvalue_reference<T>::type` is the lvalue reference type, not an rvalue reference.

For convenience, <type_traits> defines a helper template, `add_rvalue_reference_t`, that aliases the `type` member of `add_rvalue_reference`.

Example

This code example uses static_assert to show how rvalue reference types are created by using `add_rvalue_reference` and `add_rvalue_reference_t`, and how the result of `add_rvalue_reference` on an lvalue reference type is not an rvalue reference, but collapses to the lvalue reference type.

C++

```
// ex_add_rvalue_reference.cpp
// Build by using: cl /EHsc /W4 ex_add_rvalue_reference.cpp
#include <type_traits>
#include <iostream>
#include <string>

using namespace std;
int main()
{
    static_assert(is_same<add_rvalue_reference<string>::type,
string&&>::value,
        "Expected add_rvalue_reference_t<string> to be string&&");
    static_assert(is_same<add_rvalue_reference_t<string*>,
string*&&>::value,
        "Expected add_rvalue_reference_t<string*> to be string*&&");
    static_assert(is_same<add_rvalue_reference<string&>::type,
string&>::value,
        "Expected add_rvalue_reference_t<string&> to be string&");
    static_assert(is_same<add_rvalue_reference_t<string&&>,
string&&>::value,
        "Expected add_rvalue_reference_t<string&&> to be string&&");
    cout << "All static_assert tests of add_rvalue_reference passed." <<
endl;
    return 0;
}

/*Output:
All static_assert tests of add_rvalue_reference passed.
*/
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
add_lvalue_reference Class
is_rvalue_reference Class

add_pointer Class

Article • 08/03/2021

Makes a pointer-to-type from a specified type.

Syntax

```
C++
```

```
template <class T>
struct add_pointer;

template <class T>
using add_pointer_t = typename add_pointer<T>::type;
```

Parameters

T
The type to modify.

Remarks

The member `typedef` `type` names the same type as `remove_reference<T>::type*`. The alias `add_pointer_t` is a shortcut to access the member `typedef` `type`.

Because it is invalid to make a pointer from a reference, `add_pointer` removes the reference, if any, from the specified type before it makes a pointer-to-type. Consequently, you can use a type with `add_pointer` without being concerned about whether the type is a reference.

Example

The following example demonstrates that `add_pointer` of a type is the same as a pointer to that type.

```
C++
```

```
#include <type_traits>
#include <iostream>

int main()
```

```
{
    std::add_pointer_t<int> *p = (int **)0;

    p = p;  // to quiet "unused" warning
    std::cout << "add_pointer_t<int> == "
        << typeid(*p).name() << std::endl;

    return (0);
}
```

Output

```
add_pointer_t<int> == int *
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_pointer Class

add_volatile Class

Article • 08/03/2021

Makes a `volatile` type from the specified type.

Syntax

C++

```
template <class Ty>
struct add_volatile;

template <class T>
using add_volatile_t = typename add_volatile<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `add_volatile<T>` has a member `typedef` `type` that is *T* if *T* is a reference, a function, or a volatile-qualified type, otherwise `volatile` *T*. The alias `add_volatile_t` is a shortcut to access the member `typedef` `type`.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
{
    std::add_volatile_t<int> *p = (volatile int *)0;

    p = p;  // to quiet "unused" warning
    std::cout << "add_volatile<int> == "
        << typeid(*p).name() << std::endl;
```

```
    return (0);
}
```

Output

```
add_volatile<int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_volatile Class

aligned_storage Class

Article • 08/03/2021

Makes suitably aligned type.

Syntax

C++

```
template <std::size_t Len, std::size_t Align>
struct aligned_storage;

template <std::size_t Len, std::size_t Align =
alignment_of<max_align_t>::value>
using aligned_storage_t = typename aligned_storage<Len, Align>::type;
```

Parameters

Len
The object size.

Align
The object alignment.

Remarks

The template member typedef `type` is a synonym for a POD type with alignment *Align* and size *Len*. *Align* must be equal to `alignment_of<T>::value` for some type `T`, or to the default alignment.

Example

C++

```
#include <type_traits>
#include <iostream>

typedef std::aligned_storage<sizeof (int),
    std::alignment_of<double>::value>::type New_type;
int main()
    {
    std::cout << "alignment_of<int> == "
```

```
        << std::alignment_of<int>::value << std::endl;
    std::cout << "aligned to double == "
        << std::alignment_of<New_type>::value << std::endl;

    return (0);
    }
```

Output

```
alignment_of<int> == 4
aligned to double == 8
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
alignment_of Class

aligned_union Class

Article • 08/03/2021

Provides a POD type large enough and suitably aligned to store a union type, and the size required.

Syntax

```
C++
```

```
template <std::size_t Len, class... Types>
struct aligned_union;

template <std::size_t Len, class... Types>
using aligned_union_t = typename aligned_union<Len, Types...>::type;
```

Parameters

Len
The alignment value for the largest type in the union.

Types
The distinct types in the underlying union.

Remarks

Use the class template to get the alignment and size needed to store a union in uninitialized storage. The member typedef `type` names a POD type suitable for storage of any type listed in *Types*; the minimum size is *Len*. The static member `alignment_value` of type `std::size_t` contains the strictest alignment required of all the types listed in *Types*.

Example

The following example shows how to use `aligned_union` to allocate an aligned stack buffer to place a union.

```
C++
```

```cpp
// std__type_traits__aligned_union.cpp
#include <iostream>
#include <type_traits>

union U_type
{
    int i;
    float f;
    double d;
    U_type(float e) : f(e) {}
};

typedef std::aligned_union<16, int, float, double>::type aligned_U_type;

int main()
{
    // allocate memory for a U_type aligned on a 16-byte boundary
    aligned_U_type au;
    // do placement new in the aligned memory on the stack
    U_type* u = new (&au) U_type(1.0f);
    if (nullptr != u)
    {
        std::cout << "value of u->i is " << u->i << std::endl;
        // must clean up placement objects manually!
        u->~U_type();
    }
}
```

Output

```
value of u->i is 1065353216
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
alignment_of Class

alignment_of Class

Article • 08/03/2021

Gets alignment of the specified type. This struct is implemented in terms of alignof. Use `alignof` directly when you simply need to query an alignment value. Use `alignment_of` when you need an integral constant, for example when doing tag dispatch.

Syntax

```
C++
```

```
template <class Ty>
struct alignment_of;
```

Parameters

Ty
The type to query.

Remarks

The type query holds the value of the alignment of the type *Ty*.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
aligned_storage Class

common_type Class

Article • 08/03/2021

Determines the common type of one or more types.

Syntax

```
C++
```

```
template <class... T>
struct common_type;

template <class T>
struct common_type<T> {
    typedef typename decay<T>::type type;
};

template <class T, class U>
struct common_type<T, U> {
    typedef typename decay<decltype(true declval<T>() :
    declval<U>())>::type type;
};

template <class T, class U, class... V>
struct common_type<T, U, V...> {
    typedef typename common_type<typename common_type<T, U>::type,
V...>::type type;
};
```

Parameters

List of types that are either complete types or void.

Remarks

The `type` member is the common type to which all types in the parameter list can be converted.

Example

The following program demonstrates some correct usage scenarios and tests for results.

```
C++
```

```
// Compile using cl.exe /EHsc
// common_type sample
#include <iostream>
#include <type_traits>

struct Base {};
struct Derived : Base {};

int main()
{
    typedef std::common_type<unsigned char, short, int>::type NumericType;
    typedef std::common_type<float, double>::type FloatType;
    typedef std::common_type<const int, volatile int>::type ModifiedIntType;
    typedef std::common_type<Base, Derived>::type ClassType;

    std::cout << std::boolalpha;
    std::cout << "Test for typedefs of common_type int" << std::endl;
    std::cout << "NumericType: "     << std::is_same<int,
NumericType>::value << std::endl;
    std::cout << "FloatType: "       << std::is_same<int, FloatType>::value
<< std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<int,
ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: "       << std::is_same<int, ClassType>::value
<< std::endl;
    std::cout << "---------------------------" << std::endl;
    std::cout << "Test for typedefs of common_type double" << std::endl;
    std::cout << "NumericType: "     << std::is_same<double,
NumericType>::value << std::endl;
    std::cout << "FloatType: "       << std::is_same<double,
FloatType>::value << std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<double,
ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: "       << std::is_same<double,
ClassType>::value << std::endl;
    std::cout << "---------------------------" << std::endl;
    std::cout << "Test for typedefs of common_type Base" << std::endl;
    std::cout << "NumericType: "     << std::is_same<Base,
NumericType>::value << std::endl;
    std::cout << "FloatType: "       << std::is_same<Base, FloatType>::value
<< std::endl;
    std::cout << "ModifiedIntType: " << std::is_same<Base,
ModifiedIntType>::value << std::endl;
    std::cout << "ClassType: "       << std::is_same<Base, ClassType>::value
<< std::endl;

    return 0;
}
```

Output

```
```

Output

```
Test for typedefs of common_type int
NumericType: true
FloatType: false
ModifiedIntType: true
ClassType: false
---------------------------
Test for typedefs of common_type double
NumericType: false
FloatType: true
ModifiedIntType: false
ClassType: false
---------------------------
Test for typedefs of common_type Base
NumericType: false
FloatType: false
ModifiedIntType: false
ClassType: true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

conditional Class

Article • 08/03/2021

Selects one of two types, depending on the specified condition.

Syntax

C++

```
template <bool B, class T1, class T2>
struct conditional;

template <bool _Test, class _T1, class _T2>
using conditional_t = typename conditional<_Test, _T1, _T2>::type;
```

Parameters

B
The value that determines the selected type.

T1
The type result when B is true.

T2
The type result when B is false.

Remarks

The template member typedef `conditional<B, T1, T2>::type` evaluates to *T1* when *B* evaluates to `true`, and evaluates to *T2* when *B* evaluates to `false`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

decay Class

Article • 08/03/2021

Produces the type as passed by value. Makes the type non-reference, non-const, non-volatile, or makes a pointer to the type from a function or an array type.

Syntax

```
C++
```

```
template <class T>
struct decay;

template <class T>
using decay_t = typename decay<T>::type;
```

Parameters

T
The type to modify.

Remarks

Use the decay template to produce the resulting type as if the type was passed by value as an argument. The class template member typedef `type` holds a modified type that is defined in the following stages:

- The type `U` is defined as `remove_reference<T>::type`.
- If `is_array<U>::value` is true, the modified type `type` is `remove_extent<U>::type *`.
- Otherwise, if `is_function<U>::value` is true, the modified type `type` is `add_pointer<U>::type`.
- Otherwise, the modified type `type` is `remove_cv<U>::type`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

`enable_if` Class

Article • 03/04/2022

Conditionally makes an instance of a type for SFINAE overload resolution. The nested typedef `enable_if<Condition,Type>::type` exists—and is a synonym for `Type`—if and only if `Condition` is **`true`**.

Syntax

```
C++
```

```
template <bool B, class T = void>
struct enable_if;
```

Parameters

`B`

The value that determines the existence of the resulting type.

`T`

The type to instantiate if *`B`* is `true`.

Remarks

If *`B`* is `true`, `enable_if<B, T>` has a nested typedef named "type" that's a synonym for *`T`*.

If *`B`* is `false`, `enable_if<B, T>` doesn't have a nested typedef named "type".

This alias template is provided:

```
C++
```

```
template <bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

In C++, substitution failure of template parameters isn't an error in itself—this is referred to as *SFINAE* (substitution failure isn't an error). Typically, `enable_if` is used to remove candidates from overload resolution—that is, it culls the overload set—so that

one definition can be rejected in favor of another. This conforms to SFINAE behavior. For more information about SFINAE, see Substitution failure isn't an error on Wikipedia.

Here are four example scenarios:

- Scenario 1: Wrapping the return type of a function:

```
C++
```

```
    template <your_stuff>
typename enable_if<your_condition, your_return_type>::type
    yourfunction(args) {// ...
}
// The alias template makes it more concise:
    template <your_stuff>
enable_if_t<your_condition, your_return_type>
yourfunction(args) {// ...
}
```

- Scenario 2: Adding a function parameter that has a default argument:

```
C++
```

```
    template <your_stuff>
your_return_type_if_present
    yourfunction(args, enable_if_t<your condition, FOO> = BAR) {// ...
}
```

- Scenario 3: Adding a template parameter that has a default argument:

```
C++
```

```
    template <your_stuff, typename Dummy = enable_if_t<your_condition>>
rest_of_function_declaration_goes_here
```

- Scenario 4: If your function has a non-templated argument, you can wrap its type:

```
C++
```

```
    template <typename T>
void your_function(const T& t,
    enable_if_t<is_something<T>::value, const string&>
s) {// ...
}
```

Scenario 1 doesn't work with constructors and conversion operators because they don't have return types.

Scenario 2 leaves the parameter unnamed. You could say `::type Dummy = BAR`, but the name `Dummy` is irrelevant, and giving it a name is likely to trigger an "unreferenced parameter" warning. You have to choose a `FOO` function parameter type and `BAR` default argument. You could say `int` and `0`, but then users of your code could accidentally pass to the function an extra integer that would be ignored. Instead, we recommend that you use `void **` and either `0` or `nullptr` because almost nothing is convertible to `void **`:

```
C++

template <your_stuff>
your_return_type_if_present
yourfunction(args, typename enable_if<your_condition, void **>::type =
nullptr) {// ...
}
```

Scenario 2 also works for ordinary constructors. However, it doesn't work for conversion operators because they can't take extra parameters. It also doesn't work for variadic constructors because adding extra parameters makes the function parameter pack a non-deduced context and thereby defeats the purpose of `enable_if`.

Scenario 3 uses the name `Dummy`, but it's optional. Just " `typename = typename`" would work, but if you think that looks weird, you can use a "dummy" name—just don't use one that might also be used in the function definition. If you don't give a type to `enable_if`, it defaults to void, and that's perfectly reasonable because you don't care what `Dummy` is. This works for everything, including conversion operators and variadic constructors.

Scenario 4 works in constructors that don't have return types, and thereby solves the wrapping limitation of Scenario 1. However, Scenario 4 is limited to non-templated function arguments, which aren't always available. (Using Scenario 4 on a templated function argument prevents template argument deduction from working on it.)

`enable_if` is powerful, but also dangerous if it's misused. Because its purpose is to make candidates vanish before overload resolution, when it's misused, its effects can be very confusing. Here are some recommendations:

- Don't use `enable_if` to select between implementations at compile-time. Don't ever write one `enable_if` for `CONDITION` and another for `!CONDITION`. Instead, use a *tag dispatch* pattern—for example, an algorithm that selects implementations depending on the strengths of the iterators they're given.
- Don't use `enable_if` to enforce requirements. If you want to validate template parameters, and if the validation fails, cause an error instead of selecting another

implementation, use static_assert.

- Use `enable_if` when you have an overload set that makes otherwise good code ambiguous. Most often, this occurs in implicitly converting constructors.

Example

This example explains how the C++ Standard Library template function std::make_pair() takes advantage of `enable_if`.

```
C++

void func(const pair<int, int>&);

void func(const pair<string, string>&);

func(make_pair("foo", "bar"));
```

In this example, `make_pair("foo", "bar")` returns `pair<const char *, const char *>`. Overload resolution has to determine which `func()` you want. `pair<A, B>` has an implicitly converting constructor from `pair<X, Y>`. This isn't new—it was in C++98. However, in C++98/03, the implicitly converting constructor's signature always exists, even if it's `pair<int, int>(const pair<const char *, const char *>&)`. Overload resolution doesn't care that an attempt to instantiate that constructor explodes horribly because `const char *` isn't implicitly convertible to **`int`**; it's only looking at signatures, before function definitions are instantiated. Therefore, the example code is ambiguous, because signatures exist to convert `pair<const char *, const char *>` to both `pair<int, int>` and `pair<string, string>`.

C++11 solved this ambiguity by using `enable_if` to make sure `pair<A, B>(const pair<X, Y>&)` exists **only** when `const X&` is implicitly convertible to `A` and `const Y&` is implicitly convertible to `B`. This allows overload resolution to determine that `pair<const char *, const char *>` isn't convertible to `pair<int, int>` and that the overload that takes `pair<string, string>` is viable.

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

<type_traits>

extent Class

Article • 08/03/2021

Gets an array dimension.

Syntax

C++

```
template <class Ty, unsigned I = 0>
struct extent;
```

Parameters

Ty
The type to query.

I
The array bound to query.

Remarks

If *Ty* is an array type that has at least *I* dimensions, the type query holds the number of elements in the dimension specified by *I*. If *Ty* is not an array type or its rank is less than *I*, or if *I* is zero and *Ty* is of type "array of unknown bound of `U`", the type query holds the value 0.

Example

C++

```
// std__type_traits__extent.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
    {
    std::cout << "extent 0 == "
        << std::extent<int[5][10]>::value << std::endl;
    std::cout << "extent 1 == "
        << std::extent<int[5][10], 1>::value << std::endl;
```

```
    return (0);
    }
```

Output

```
extent 0 == 5
extent 1 == 10
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_all_extents Class
remove_extent Class

integer_sequence Class

Article • 08/03/2021

Represents an integer sequence. Can be used to deduce and expand parameter packs in variadic types such as std::tuple<T...> that are passed as arguments to a function.

Syntax

```
C++
```

```
template <class T, T... Vals>
struct integer_sequence
```

Parameters

T
The type of the values; must be an integral type: bool, char, char16_t, char32_t, wchar_t, or signed or unsigned integer types.

Vals
A non-type parameter pack that represents a sequence of values of integral type T.

Members

Name	Description
`static size_t size() noexcept`	The number of elements in the sequence.
`typedef T value_type`	The type of each element in the sequence. Must be an integral type.

Remarks

A parameter pack that is passed directly to a function can be unpacked without any special library helpers. When a parameter pack is part of a type that is passed to a function, and you need indices to access the elements, then the easiest way to unpack it is to use `integer_sequence` and its related type aliases `make_integer_sequence`, `index_sequence`, `make_index_sequence`, and `index_sequence_for`.

Example

The following example is based on the original proposal N3658. It shows how to use an `integer_sequence` to create a `std::tuple` from a `std::array<T,N>`, and how to use an `integer_sequence` to get at the tuple members.

In the `a2t` function, an `index_sequence` is an alias of `integer_sequence` based on the `size_t` integral type. `make_index_sequence` is an alias that at compile time creates a zero-based `index_sequence` with the same number of elements as the array that is passed in by the caller. `a2t` passes the `index_sequence` by value to `a2t_` , where the expression `a[I]...` unpacks `I`, and then the elements are being fed to `make_tuple` which consumes them as individual arguments. For example, if the sequence contains three elements, then `make_tuple` is called as make_tuple(a[0], a[1], a[2]). The array elements themselves can of course be any type.

The apply function accepts a std::tuple, and produces an `integer_sequence` by using the `tuple_size` helper class. Note that std::decay_t is necessary because tuple_size does not work with reference types. The `apply_` function unpacks the tuple members and forwards them as separate arguments to a function call. In this example the function is a simple lambda expression that prints out the values.

C++

```
#include <stddef.h>
#include <iostream>
#include <tuple>
#include <utility>
#include <array>
#include <string>

using namespace std;

// Create a tuple from the array and the index_sequence
template<typename Array, size_t... I>
auto a2t_(const Array& a, index_sequence<I...>)
{
    return make_tuple(a[I]...);
}

// Create an index sequence for the array, and pass it to the
// implementation function a2t_
template<typename T, size_t N>
auto a2t(const array<T, N>& a)
{
    return a2t_(a, make_index_sequence<N>());
}
```

```
// Call function F with the tuple members as separate arguments.
template<typename F, typename Tuple = tuple<T...>, size_t... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return forward<F>(f)(get<I>(forward<Tuple>(args))...);
}

// Create an index_sequence for the tuple, and pass it with the
// function object and the tuple to the implementation function apply_
template<typename F, typename Tuple = tuple<T...>>
decltype(auto) apply(F&& f, Tuple&& args)
{
    using Indices = make_index_sequence<tuple_size<decay_t<Tuple>>::value >;
    return apply_(forward<F>(f), forward<Tuple>(args), Indices());
}

int main()
{
    const array<string, 3> arr { "Hello", "from", "C++14" };

    //Create a tuple given a array
    auto tup = a2t(arr);

    // Extract the tuple elements
    apply([](const string& a, const string& b, const string& c) {cout << a
<< " " << b << " " << c << endl; }, tup);

    char c;
    cin >> c;
}
```

To make an `index_sequence` for a parameter pack, use `index_sequence_for`<T...> which is an alias for `make_index_sequence`<sizeof...(T)>

Requirements

Header: <type_traits>

Namepace: std

See also

Ellipsis and Variadic Templates

integral_constant Class, bool_constant Class

Article • 08/03/2021

Makes an integral constant from a type and value.

Syntax

C++

```
template<class T, T v>
struct integral_constant {
   static constexpr T value = v;
   typedef T value_type;
   typedef integral_constant<T, v> type;
   constexpr operator value_type() const noexcept;
   constexpr value_type operator()() const noexcept;
   };
```

Parameters

T
The type of the constant.

v
The value of the constant.

Remarks

The `integral_constant` class template, when specialized with an integral type *T* and a value *v* of that type, represents an object that holds a constant of that integral type with the specified value. The member named `type` is an alias for the generated template specialization type, and the `value` member holds the value *v* used to create the specialization.

The `bool_constant` class template is an explicit partial specialization of `integral_constant` that uses `bool` as the *T* argument.

Example

C++

```
// std__type_traits__integral_constant.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
    {
    std::cout << "integral_constant<int, 5> == "
        << std::integral_constant<int, 5>::value << std::endl;
    std::cout << "integral_constant<bool, false> == " << std::boolalpha
        << std::integral_constant<bool, false>::value << std::endl;

    return (0);
    }
```

Output

```
integral_constant<int, 5> == 5
integral_constant<bool, false> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
false_type
true_type

invoke_result Class

Article • 08/03/2021

Determines the return type of the callable type that takes the specified argument types at compile time. Added in C++17.

Syntax

```
C++
```

```
template <class Callable, class... Args>
   struct invoke_result<Callable(Args...)>;

// Helper type
template<class Callable, class... Args>
   using invoke_result_t = typename invoke_result<Callable, Args...>::type;
```

Parameters

Callable
The callable type to query.

Args
The types of the argument list to the callable type to query.

Remarks

Use this template to determine the result type of *Callable*(*Args*...) at compile time, where *Callable* and all types in *Args* are any complete type, an array of unknown bound, or a possibly cv-qualified `void`. The `type` member of the class template names the return type of *Callable* when invoked using the arguments *Args*.... The `type` member is only defined if *Callable* can be called when invoked using the arguments *Args*... in an unevaluated context. Otherwise, the class template has no member `type`, which allows SFINAE tests on a particular set of argument types at compile time.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
invoke

is_abstract Class

Article • 08/03/2021

Tests if type is abstract class.

Syntax

C++

```
template <class Ty>
struct is_abstract;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has at least one pure virtual function, otherwise it holds false.

Example

C++

```
// std__type_traits__is_abstract.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct abstract
    {
    virtual int val() = 0;
    };

int main()
    {
```

```
    std::cout << "is_abstract<trivial> == " << std::boolalpha
        << std::is_abstract<trivial>::value << std::endl;
    std::cout << "is_abstract<abstract> == " << std::boolalpha
        << std::is_abstract<abstract>::value << std::endl;

    return (0);
    }
```

Output

```
is_abstract<trivial> == false
is_abstract<abstract> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_polymorphic Class

is_arithmetic Class

Article • 08/03/2021

Tests if type is arithmetic.

Syntax

C++

```
template <class Ty>
struct is_arithmetic;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an arithmetic type, that is, an integral type or a floating point type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

C++

```
// std__type_traits__is_arithmetic.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_arithmetic<trivial> == " << std::boolalpha
        << std::is_arithmetic<trivial>::value << std::endl;
    std::cout << "is_arithmetic<int> == " << std::boolalpha
```

```
        << std::is_arithmetic<int>::value << std::endl;
    std::cout << "is_arithmetic<float> == " << std::boolalpha
        << std::is_arithmetic<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_arithmetic<trivial> == false
is_arithmetic<int> == true
is_arithmetic<float> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_floating_point Class
is_integral Class

is_array Class

Article • 08/03/2021

Tests if type is array.

Syntax

C++

```
template <class Ty>
struct is_array;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an array type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_array.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_array<trivial> == " << std::boolalpha
        << std::is_array<trivial>::value << std::endl;
    std::cout << "is_array<int> == " << std::boolalpha
        << std::is_array<int>::value << std::endl;
    std::cout << "is_array<int[5]> == " << std::boolalpha
```

```
        << std::is_array<int[5]>::value << std::endl;

    return (0);
    }
```

Output

```
is_array<trivial> == false
is_array<int> == false
is_array<int[5]> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
extent Class
rank Class

is_assignable Class

Article • 08/03/2021

Tests whether a value of `From` type can be assigned to a `To` type.

Syntax

C++

```
template <class To, class From>
struct is_assignable;
```

Parameters

To
The type of the object that receives the assignment.

From
The type of the object that provides the value.

Remarks

The unevaluated expression `declval<To>() = declval<From>()` must be well-formed.

Both `From` and `To` must be complete types, **`void`**, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_base_of Class

Article • 08/03/2021

Tests whether one type is base of another.

Syntax

C++

```
template <class Base, class Derived>
struct is_base_of;
```

Parameters

Base
The base class to test for.

Derived
The derived type to test for.

Remarks

An instance of the type predicate holds true if the type *Base* is a base class of the type *Derived*, otherwise it holds false.

Example

C++

```
#include <type_traits>
#include <iostream>

struct base
    {
    int val;
    };

struct derived
    : public base
    {
    };
```

```
int main()
    {
    std::cout << "is_base_of<base, base> == " << std::boolalpha
        << std::is_base_of<base, base>::value << std::endl;
    std::cout << "is_base_of<base, derived> == " << std::boolalpha
        << std::is_base_of<base, derived>::value << std::endl;
    std::cout << "is_base_of<derived, base> == " << std::boolalpha
        << std::is_base_of<derived, base>::value << std::endl;

    return (0);
    }
```

Output

```
is_base_of<base, base> == true
is_base_of<base, derived> == true
is_base_of<derived, base> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_convertible Class

is_class Class

Article • 08/03/2021

Tests if type is a class.

Syntax

C++

```
template <class Ty>
struct is_class;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a type defined as a `class` or a `struct`, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

C++

```
// std__type_traits__is_class.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_class<trivial> == " << std::boolalpha
        << std::is_class<trivial>::value << std::endl;
    std::cout << "is_class<int> == " << std::boolalpha
        << std::is_class<int>::value << std::endl;
```

```
    return (0);
    }
```

Output

```
is_class<trivial> == true
is_class<int> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_compound Class
is_union Class

is_compound Class

Article • 08/03/2021

Tests if the specified type is not fundamental.

Syntax

C++

```
template <class Ty>
struct is_compound;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds `false` if the type of *Ty* is a fundamental type (that is, if is_fundamental<Ty> holds `true`); otherwise, it holds `true`. Thus, the predicate holds `true` if *Ty* is an array type, a function type, a pointer to `void` or an object or a function, a reference, a class, a union, an enumeration, or a pointer to non-static class member, or a *cv-qualified* form of one of them.

Example

C++

```
// std__type_traits__is_compound.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_compound<trivial> == " << std::boolalpha
```

```
        << std::is_compound<trivial>::value << std::endl;
    std::cout << "is_compound<int[]> == " << std::boolalpha
        << std::is_compound<int[]>::value << std::endl;
    std::cout << "is_compound<int()> == " << std::boolalpha
        << std::is_compound<int()>::value << std::endl;
    std::cout << "is_compound<int&> == " << std::boolalpha
        << std::is_compound<int&>::value << std::endl;
    std::cout << "is_compound<void *> == " << std::boolalpha
        << std::is_compound<void *>::value << std::endl;
    std::cout << "is_compound<int> == " << std::boolalpha
        << std::is_compound<int>::value << std::endl;

    return (0);
    }
```

Output

```
is_compound<trivial> == true
is_compound<int[]> == true
is_compound<int()> == true
is_compound<int&> == true
is_compound<void *> == true
is_compound<int> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_class Class

is_const Class

Article • 08/03/2021

Tests if type is const.

Syntax

C++

```
template <class Ty>
struct is_const;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if *Ty* is `const-qualified`.

Example

C++

```
// std__type_traits__is_const.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
{
    int val;
};

int main()
{
    std::cout << "is_const<trivial> == " << std::boolalpha
        << std::is_const<trivial>::value << std::endl;
    std::cout << "is_const<const trivial> == " << std::boolalpha
        << std::is_const<const trivial>::value << std::endl;
    std::cout << "is_const<int> == " << std::boolalpha
        << std::is_const<int>::value << std::endl;
```

```
    std::cout << "is_const<const int> == " << std::boolalpha
        << std::is_const<const int>::value << std::endl;

    return (0);
}
```

Output

```
is_const<trivial> == false
is_const<const trivial> == true
is_const<int> == false
is_const<const int> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_volatile Class

is_constructible Class

Article • 08/03/2021

Tests whether a type is constructible when the specified argument types are used.

Syntax

```
C++
```

```
template <class T, class... Args>
struct is_constructible;
```

Parameters

T
The type to query.

Args
The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is constructible by using the argument types in *Args*, otherwise it holds false. Type *T* is constructible if the variable definition `T t(std::declval<Args>()...);` is well-formed. Both *T* and all the types in *Args* must be complete types, `void`, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_convertible Class

Article • 08/03/2021

Tests if one type is convertible to another.

Syntax

C++

```
template <class From, class To>
struct is_convertible;
```

Parameters

From
The type to convert from.

Ty
The type to convert to.

Remarks

An instance of the type predicate holds true if the expression `To to = from;`, where `from` is an object of type `From`, is well-formed.

Example

C++

```
// std__type_traits__is_convertible.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_convertible<trivial, int> == " << std::boolalpha
```

```
        << std::is_convertible<trivial, int>::value << std::endl;
    std::cout << "is_convertible<trivial, trivial> == " << std::boolalpha
        << std::is_convertible<trivial, trivial>::value << std::endl;
    std::cout << "is_convertible<char, int> == " << std::boolalpha
        << std::is_convertible<char, int>::value << std::endl;

    return (0);
    }
```

Output

```
is_convertible<trivial, int> == false
is_convertible<trivial, trivial> == true
is_convertible<char, int> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_base_of Class

is_copy_assignable Class

Article • 08/03/2021

Tests whether type has can be copied on assignment.

Syntax

```
C++
```

```
template <class Ty>
struct is_copy_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy assignment operator, otherwise it holds false. Equivalent to is_assignable<Ty&, const Ty&>.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_copy_constructible Class

Article • 08/03/2021

Tests if type has a copy constructor.

Syntax

C++

```
template <class Ty>
struct is_copy_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy constructor, otherwise it holds false.

Example

C++

```
#include <type_traits>
#include <iostream>

struct Copyable
{
    int val;
};

struct NotCopyable
{
   NotCopyable(const NotCopyable&) = delete;
   int val;

};

int main()
{
```

```
    std::cout << "is_copy_constructible<Copyable> == " << std::boolalpha
        << std::is_copy_constructible<Copyable>::value << std::endl;
    std::cout << "is_copy_constructible<NotCopyable> == " << std::boolalpha
        << std::is_copy_constructible<NotCopyable>::value << std::endl;

    return (0);
}
```

Output

```
is_copy_constructible<Copyable> == true
is_copy_constructible<NotCopyable > == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_default_constructible Class

Article • 08/03/2021

Tests if a type has a default constructor.

Syntax

```
C++
```

```
template <class Ty>
struct is_default_constructible;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type that has a default constructor, otherwise it holds false. This is equivalent to the predicate `is_constructible<T>`. Type *T* must be a complete type, **`void`**, or an array of unknown bound.

Example

```
C++
```

```
#include <type_traits>
#include <iostream>

struct Simple
{
    Simple() : val(0) {}
    int val;
};

struct Simple2
{
    Simple2(int v) : val(v) {}
    int val;
};
```

```
int main()
{
    std::cout << "is_default_constructible<Simple> == " << std::boolalpha
        << std::is_default_constructible<Simple>::value << std::endl;
    std::cout << "is_default_constructible<Simple2> == " << std::boolalpha
        << std::is_default_constructible<Simple2>::value << std::endl;

    return (0);
}
```

Output

```
is_default_constructible<Simple> == true
is_default_constructible<Simple2> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_destructible Class

Article • 08/03/2021

Tests whether the type is destructible.

Syntax

```
C++
```

```
template <class T>
struct is_destructible;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, otherwise it holds false. Destructible types are reference types, object types, and types where for some type `U` equal to `remove_all_extents_t<T>` the unevaluated operand `std::declval<U&>.~U()` is well-formed. Other types, including incomplete types, `void`, and function types, are not destructible types.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_empty Class

Article • 08/03/2021

Tests if type is an empty class.

Syntax

C++

```
template <class Ty>
struct is_empty;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an empty class, otherwise it holds false.

Example

C++

```
// std__type_traits__is_empty.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct empty
    {
    };

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_empty<trivial> == " << std::boolalpha
```

```
        << std::is_empty<trivial>::value << std::endl;
    std::cout << "is_empty<empty> == " << std::boolalpha
        << std::is_empty<empty>::value << std::endl;
    std::cout << "is_empty<int> == " << std::boolalpha
        << std::is_empty<int>::value << std::endl;

    return (0);
    }
```

Output

```
is_empty<trivial> == false
is_empty<empty> == true
is_empty<int> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_enum Class

Article • 08/03/2021

Tests if type is an enumeration.

Syntax

C++

```
template <class Ty>
struct is_enum;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an enumeration type or a `cv-qualified` form of an enumeration type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_enum.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

enum color {
    red, greed, blue};

int main()
    {
    std::cout << "is_enum<trivial> == " << std::boolalpha
        << std::is_enum<trivial>::value << std::endl;
```

```
    std::cout << "is_enum<color> == " << std::boolalpha
        << std::is_enum<color>::value << std::endl;
    std::cout << "is_enum<int> == " << std::boolalpha
        << std::is_enum<int>::value << std::endl;

    return (0);
    }
```

Output

```
is_enum<trivial> == false
is_enum<color> == true
is_enum<int> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_integral Class

is_final Class

Article • 08/03/2021

Tests whether the type is a class type marked `final`.

Syntax

```
C++
```

```
template <class T>
struct is_final;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type marked `final`, otherwise it holds false. If *T* is a class type, it must be a complete type.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
final Specifier

is_floating_point Class

Article • 08/03/2021

Tests if type is floating point.

Syntax

C++

```
template <class Ty>
struct is_floating_point;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a floating point type or a `cv-qualified` form of a floating point type, otherwise it holds false.

A floating point type is one of **`float`**, **`double`**, or **`long double`**.

Example

C++

```
// std__type_traits__is_floating_point.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_floating_point<trivial> == " << std::boolalpha
        << std::is_floating_point<trivial>::value << std::endl;
```

```
    std::cout << "is_floating_point<int> == " << std::boolalpha
        << std::is_floating_point<int>::value << std::endl;
    std::cout << "is_floating_point<float> == " << std::boolalpha
        << std::is_floating_point<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_floating_point<trivial> == false
is_floating_point<int> == false
is_floating_point<float> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_integral Class

is_function Class

Article • 08/03/2021

Tests if type is a function type.

Syntax

C++

```
template <class Ty>
struct is_function;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a function type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_function.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct functional
    {
    int f();
    };

int main()
    {
```

```
    std::cout << "is_function<trivial> == " << std::boolalpha
        << std::is_function<trivial>::value << std::endl;
    std::cout << "is_function<functional> == " << std::boolalpha
        << std::is_function<functional>::value << std::endl;
    std::cout << "is_function<float()> == " << std::boolalpha
        << std::is_function<float()>::value << std::endl;

    return (0);
    }
```

Output

```
is_function<trivial> == false
is_function<functional> == false
is_function<float()> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_object Class

is_fundamental Class

Article • 08/03/2021

Tests if type is void or arithmetic.

Syntax

C++

```
template <class Ty>
struct is_fundamental;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a fundamental type, that is, `void`, an integral type, an floating point type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

C++

```
// std__type_traits__is_fundamental.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_fundamental<trivial> == " << std::boolalpha
        << std::is_fundamental<trivial>::value << std::endl;
    std::cout << "is_fundamental<int> == " << std::boolalpha
```

```
        << std::is_fundamental<int>::value << std::endl;
    std::cout << "is_fundamental<const float> == " << std::boolalpha
        << std::is_fundamental<const float>::value << std::endl;
    std::cout << "is_fundamental<void> == " << std::boolalpha
        << std::is_fundamental<void>::value << std::endl;

    return (0);
    }
```

Output

```
is_fundamental<trivial> == false
is_fundamental<int> == true
is_fundamental<const float> == true
is_fundamental<void> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_compound Class

is_integral Class

Article • 08/03/2021

Tests if type is integral.

Syntax

```
C++
```

```
template <class Ty>
struct is_integral;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is one of the integral types, or a `cv-qualified` form of one of the integral types, otherwise it holds false.

An integral type is one of **`bool`**, **`char`**, **`unsigned char`**, **`signed char`**, **`wchar_t`**, **`short`**, **`unsigned short`**, **`int`**, **`unsigned int`**, **`long`**, and **`unsigned long`**. In addition, with compilers that provide them, an integral type can be one of **`long long`**, **`unsigned long long`**, **`__int64`**, and **unsigned __int64**.

Example

```
C++
```

```
// std__type_traits__is_integral.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };
```

```
int main()
    {
    std::cout << "is_integral<trivial> == " << std::boolalpha
        << std::is_integral<trivial>::value << std::endl;
    std::cout << "is_integral<int> == " << std::boolalpha
        << std::is_integral<int>::value << std::endl;
    std::cout << "is_integral<float> == " << std::boolalpha
        << std::is_integral<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_integral<trivial> == false
is_integral<int> == true
is_integral<float> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_enum Class
is_floating_point Class

is_invocable, is_invocable_r, is_nothrow_invocable, is_nothrow_invocable_r classes

Article • 08/03/2021

These templates determine if a type can be invoked with the specified argument types. `is_invocable_r` and `is_nothrow_invocable_r` also determine if the result of the invocation is convertible to a specific type. `is_nothrow_invocable` and `is_nothrow_invocable_r` also determine if the invocation is known not to throw exceptions. Added in C++17.

Syntax

C++

```
template <class Callable, class... Args>
struct is_invocable;

template <class Convertible, class Callable, class... Args>
struct is_invocable_r;

template <class Callable, class... Args>
struct is_nothrow_invocable;

template <class Convertible, class Callable, class... Args>
struct is_nothrow_invocable_r;

// Helper templates
template <class Callable, class... Args>
inline constexpr bool is_invocable_v =
    std::is_invocable<Callable, Args...>::value;

template <class Convertible, class Callable, class... Args>
inline constexpr bool is_invocable_r_v =
    std::is_invocable_r<Convertible, Callable, Args...>::value;

template <class Callable, class... Args>
inline constexpr bool is_nothrow_invocable_v =
    std::is_nothrow_invocable<Callable, Args...>::value;

template <class Convertible, class Callable, class... Args>
inline constexpr bool is_nothrow_invocable_r_v =
    std::is_nothrow_invocable_r<Convertible, Callable, Args...>::value;
```

Parameters

Callable
The callable type to query.

Args
The argument types to query.

Convertible
The type the result of *Callable* must be convertible to.

Remarks

The `is_invocable` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context.

The `is_invocable_r` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context to produce a result type convertible to *Convertible*.

The `is_nothrow_invocable` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context, and that such a call is known not to throw an exception.

The `is_nothrow_invocable_r` type predicate holds true if the callable type *Callable* can be invoked using the arguments *Args* in an unevaluated context to produce a result type convertible to *Convertible*, and that such a call is known not to throw an exception.

Each of the types *Convertible*, *Callable*, and the types in the parameter pack *Args* must be a complete type, an array of unknown bound, or a possibly cv-qualified `void`. Otherwise, the behavior of the predicate is undefined.

Example

C++

```
// std__type_traits__is_invocable.cpp
// compile using: cl /EHsc /std:c++17 std__type_traits__is_invocable.cpp
#include <type_traits>

auto test1(int) noexcept -> int (*)()
{
    return nullptr;
}
```

```
auto test2(int) -> int (*)()
{
    return nullptr;
}

int main()
{
    static_assert( std::is_invocable<decltype(test1), short>::value );

    static_assert( std::is_invocable_r<int(*)(), decltype(test1),
int>::value );
    static_assert( std::is_invocable_r<long(*)(), decltype(test1),
int>::value ); // fails

    static_assert( std::is_nothrow_invocable<decltype(test1), int>::value );
    static_assert( std::is_nothrow_invocable<decltype(test2), int>::value );
// fails

    static_assert( std::is_nothrow_invocable_r<int(*)(), decltype(test1),
int>::value );
    static_assert( std::is_nothrow_invocable_r<int(*)(), decltype(test2),
int>::value ); // fails
}
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
invoke

is_literal_type Class

Article • 08/03/2021

Tests whether a type can be used as a `constexpr` variable or be constructed, used by, or returned from `constexpr` functions.

Syntax

```
C++
```

```
template <class T>
struct is_literal_type;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a *literal type*, otherwise it holds false. A literal type is either `void`, a scalar type, a reference type, an array of literal type, or a literal class type. A literal class type is a class type that has a trivial destructor, is either an aggregate type or has at least one non-move, non-copy `constexpr` constructor, and all of its base classes and non-static data members are non-volatile literal types. While the type of a literal is always a literal type, the concept of a literal type includes anything that the compiler can evaluate as a `constexpr` at compile time.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_lvalue_reference Class

Article • 08/03/2021

Tests if type is an lvalue reference.

Syntax

```
C++
```

```
template <class Ty>
struct is_lvalue_reference;
```

Parameters

Ty
The type to query.

Remarks

An instance of this type predicate holds true if the type *Ty* is a reference to an object or to a function, otherwise it holds false. Note that *Ty* may not be an rvalue reference. For more information about rvalues, see Rvalue Reference Declarator: &&.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
Lvalues and Rvalues

is_member_function_pointer Class

Article • 08/03/2021

Tests if type is a pointer to member function.

Syntax

C++

```
template <class Ty>
struct is_member_function_pointer;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member function or a `cv-qualified` pointer to member function, otherwise it holds false.

Example

C++

```
// std__type_traits__is_member_function_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct functional
    {
    int f();
    };

int main()
    {
```

```
    std::cout << "is_member_function_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_function_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_function_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_function_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_function_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_function_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
    }
```

Output

```
is_member_function_pointer<trivial *> == false
is_member_function_pointer<int trivial::*> == false
is_member_function_pointer<int (functional::*)()> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_member_pointer Class

is_member_object_pointer Class

Article • 08/03/2021

Tests if type is a pointer to member object.

Syntax

C++

```
template <class Ty>
struct is_member_object_pointer;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member object or a `cv-qualified` pointer to member object, otherwise it holds false. Note that `is_member_object_pointer` holds false if *Ty* is a pointer to member function.

Example

C++

```
// std__type_traits__is_member_object_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct functional
    {
    int f();
    };
```

```
int main()
    {
    std::cout << "is_member_object_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_object_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_object_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_object_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_object_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_object_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
    }
```

Output

```
is_member_object_pointer<trivial *> == false
is_member_object_pointer<int trivial::*> == true
is_member_object_pointer<int (functional::*)()> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_member_pointer Class

is_member_pointer Class

Article • 08/03/2021

Tests if type is a pointer to member.

Syntax

C++

```
template <class Ty>
struct is_member_pointer;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to member function or a pointer to member object, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

C++

```
// std__type_traits__is_member_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct functional
    {
    int f();
    };
```

```
int main()
    {
    std::cout << "is_member_pointer<trivial *> == "
        << std::boolalpha
        << std::is_member_pointer<trivial *>::value
        << std::endl;
    std::cout << "is_member_pointer<int trivial::*> == "
        << std::boolalpha
        << std::is_member_pointer<int trivial::*>::value
        << std::endl;
    std::cout << "is_member_pointer<int (functional::*)()> == "
        << std::boolalpha
        << std::is_member_pointer<int (functional::*)()>::value
        << std::endl;

    return (0);
    }
```

Output

```
is_member_pointer<trivial *> == false
is_member_pointer<int trivial::*> == true
is_member_pointer<int (functional::*)()> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_member_function_pointer Class
is_member_object_pointer Class
is_pointer Class

is_move_assignable Class

Article • 08/03/2021

Tests if the type can be move assigned.

Syntax

```
C++
```

```
template <class T>
struct is_move_assignable;
```

Parameters

T
The type to query.

Remarks

A type is move assignable if an rvalue reference to the type can be assigned to a reference to the type. The type predicate is equivalent to `is_assignable<T&, T&&>`. Move assignable types include referenceable scalar types and class types that have either compiler-generated or user-defined move assignment operators.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_move_constructible class

Article • 08/03/2021

Tests whether the type can be constructed by using a move operation.

Syntax

```
C++
```

```
template <class T>
struct is_move_constructible;
```

Parameters

T
The type to be evaluated.

Remarks

A type predicate that evaluates to `true` if the type *T* can be constructed by using a move operation. This predicate is equivalent to `is_constructible<T, T&&>`. A type *T* that doesn't have a move constructor, but does have a copy constructor that accepts a `const T&` argument, satisfies `std::is_move_constructible`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_assignable Class

Article • 08/03/2021

Tests whether a value of *From* type can be assigned to *To* type and the assignment is known not to throw.

Syntax

```
C++
```

```
template <class To, class From>
struct is_nothrow_assignable;
```

Parameters

To
The type of the object that receives the assignment.

From
The type of the object that provides the value.

Remarks

The expression `declval<To>() = declval<From>()` must be well-formed and must be known to the compiler not to throw. Both *From* and *To* must be complete types, `void`, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_constructible Class

Article • 08/03/2021

Tests whether a type is constructible and is known not to throw when the specified argument types are used.

Syntax

```
C++
```

```
template <class T, class... Args>
struct is_nothrow_constructible;
```

Parameters

T
The type to query.

Args
The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is constructible by using the argument types in *Args*, and the constructor is known by the compiler not to throw; otherwise it holds false. Type *T* is constructible if the variable definition `T t(std::declval<Args>()...);` is well-formed. Both *T* and all the types in *Args* must be complete types, `void`, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_copy_assignable Class

Article • 08/03/2021

Tests whether type has a copy assignment operator that is known to the compiler not to throw.

Syntax

```
C++
```

```
template <class T>
struct is_nothrow_copy_assignable;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true for a referenceable type *T* where `is_nothrow_assignable<T&, const T&>` holds true; otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_nothrow_assignable Class

is_nothrow_copy_constructible Class

Article • 08/03/2021

Tests whether type has a `nothrow` copy constructor.

Syntax

C++

```
template <class Ty>
struct is_nothrow_copy_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow copy constructor, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_default_constructible Class

Article • 08/03/2021

Tests whether type has a non-throwing default constructor.

Syntax

```
C++
```

```
template <class Ty>
struct is_nothrow_default_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow default constructor, otherwise it holds false. An instance of the type predicate is equivalent to `is_nothrow_constructible<Ty>`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_destructible Class

Article • 08/03/2021

Tests whether the type is destructible and the destructor is known to the compiler not to throw.

Syntax

```
C++

template <class T>
struct is_nothrow_destructible;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, and the destructor is known to the compiler not to throw. Otherwise, it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_move_assignable Class

Article • 08/03/2021

Tests whether type has a `nothrow` move assignment operator.

Syntax

C++

```
template <class Ty>
struct is_nothrow_move_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow move assignment operator, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_nothrow_move_constructible Class

Article • 08/03/2021

Tests whether type has a `nothrow` move constructor.

Syntax

C++

```
template <class Ty>
struct is_nothrow_move_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow move constructor, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_null_pointer Class

Article • 08/03/2021

Tests if type is std::nullptr_t.

Syntax

```
C++
```

```
template <class T>
struct is_null_pointer;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is `std::nullptr_t`, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_object Class

Article • 08/03/2021

Tests if type is an object type.

Syntax

C++

```
template <class Ty>
struct is_object;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds false if the type *Ty* is a reference type, a function type, or void, or a `cv-qualified` form of one of them, otherwise holds true.

Example

C++

```
// std__type_traits__is_object.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct functional
    {
    int f();
    };

int main()
    {
```

```
    std::cout << "is_object<trivial> == " << std::boolalpha
        << std::is_object<trivial>::value << std::endl;
    std::cout << "is_object<functional> == " << std::boolalpha
        << std::is_object<functional>::value << std::endl;
    std::cout << "is_object<trivial&> == " << std::boolalpha
        << std::is_object<trivial&>::value << std::endl;
    std::cout << "is_object<float()> == " << std::boolalpha
        << std::is_object<float()>::value << std::endl;
    std::cout << "is_object<void> == " << std::boolalpha
        << std::is_object<void>::value << std::endl;

    return (0);
    }
```

Output

```
is_object<trivial> == true
is_object<functional> == true
is_object<trivial&> == false
is_object<float()> == false
is_object<void> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_function Class

is_pod Class

Article • 08/03/2021

Tests if type is POD.

Syntax

```
C++
```

```
template <class T>
struct is_pod;
```

Parameters

T
The type to query.

Remarks

`is_pod<T>::value` is **`true`** if the type *T* is Plain Old Data (POD). Otherwise it is **`false`**.

Arithmetic types, enumeration types, pointer types, and pointer to member types are POD.

A cv-qualified version of a POD type is itself a POD type.

An array of POD is itself POD.

A struct or union, all of whose non-static data members are POD, is itself POD if it has:

- No user-declared constructors.
- No private or protected non-static data members.
- No base classes.
- No virtual functions.
- No non-static data members of reference type.
- No user-defined copy assignment operator.
- No user-defined destructor.

Therefore, you can recursively build POD structs and arrays that contain POD structs and arrays.

Example

C++

```
// std__type_traits__is_pod.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial {
    int val;
};

struct throws {
    throws() {}  // User-declared ctor, so not POD

    int val;
};

int main() {
    std::cout << "is_pod<trivial> == " << std::boolalpha
        << std::is_pod<trivial>::value << std::endl;
    std::cout << "is_pod<int> == " << std::boolalpha
        << std::is_pod<int>::value << std::endl;
    std::cout << "is_pod<throws> == " << std::boolalpha
        << std::is_pod<throws>::value << std::endl;

    return (0);
}
```

Output

```
is_pod<trivial> == true
is_pod<int> == true
is_pod<throws> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_pointer Class

Article • 08/03/2021

Tests if type is a pointer.

Syntax

C++

```
template <class Ty>
struct is_pointer;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a pointer to `void`, a pointer to an object, or a pointer to a function, or a `cv-qualified` form of one of them, otherwise it holds false. Note that `is_pointer` holds false if *Ty* is a pointer to member or a pointer to member function.

Example

C++

```
// std__type_traits__is_pointer.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_pointer<trivial> == " << std::boolalpha
        << std::is_pointer<trivial>::value << std::endl;
```

```
    std::cout << "is_pointer<int trivial::*> == " << std::boolalpha
        << std::is_pointer<int trivial::*>::value << std::endl;
    std::cout << "is_pointer<trivial *> == " << std::boolalpha
        << std::is_pointer<trivial *>::value << std::endl;
    std::cout << "is_pointer<int> == " << std::boolalpha
        << std::is_pointer<int>::value << std::endl;
    std::cout << "is_pointer<int *> == " << std::boolalpha
        << std::is_pointer<int *>::value << std::endl;

    return (0);
    }
```

Output

```
is_pointer<trivial> == false
is_pointer<int trivial::*> == false
is_pointer<trivial *> == true
is_pointer<int> == false
is_pointer<int *> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_member_pointer Class
is_reference Class

is_polymorphic Class

Article • 08/03/2021

Tests if type has a virtual function.

Syntax

C++

```
template <class Ty>
struct is_polymorphic;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that declares or inherits a virtual function, otherwise it holds false.

Example

C++

```
// std__type_traits__is_polymorphic.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

struct throws
    {
    throws() throw(int)
        {
        }

    throws(const throws&) throw(int)
```

```
        {
        }

    throws& operator=(const throws&) throw(int)
        {
        }

    virtual ~throws()
        {
        }

    int val;
    };

int main()
    {
    std::cout << "is_polymorphic<trivial> == " << std::boolalpha
        << std::is_polymorphic<trivial>::value << std::endl;
    std::cout << "is_polymorphic<throws> == " << std::boolalpha
        << std::is_polymorphic<throws>::value << std::endl;

    return (0);
    }
```

Output

```
is_polymorphic<trivial> == false
is_polymorphic<throws> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_abstract Class

is_reference Class

Article • 08/03/2021

Tests if type is a reference.

Syntax

C++

```
template <class Ty>
struct is_reference;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a reference to an object or to a function, otherwise it holds false.

Example

C++

```
// std__type_traits__is_reference.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_reference<trivial> == " << std::boolalpha
        << std::is_reference<trivial>::value << std::endl;
    std::cout << "is_reference<trivial&> == " << std::boolalpha
        << std::is_reference<trivial&>::value << std::endl;
    std::cout << "is_reference<int()> == " << std::boolalpha
```

```
        << std::is_reference<int()>::value << std::endl;
    std::cout << "is_reference<int(&)()> == " << std::boolalpha
        << std::is_reference<int(&)()>::value << std::endl;

    return (0);
    }
```

Output

```
is_reference<trivial> == false
is_reference<trivial&> == true
is_reference<int()> == false
is_reference<int(&)()> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_pointer Class

is_rvalue_reference Class

Article • 08/03/2021

Tests if type is an rvalue reference.

Syntax

```
C++
```

```
template <class Ty>
struct is_rvalue_reference;
```

Parameters

Ty
The type to query.

Remarks

An instance of this type predicate holds true if the type *Ty* is an rvalue reference.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
Lvalues and Rvalues

is_same Class

Article • 08/03/2021

Tests if two types are the same.

Syntax

C++

```
template <class Ty1, class Ty2>
struct is_same;
```

Parameters

Ty1
The first type to query.

Ty2
The second type to query.

Remarks

An instance of the type predicate holds true if the types *Ty1* and *Ty2* are the same type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_same.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct base
    {
    int val;
    };

struct derived
    : public base
    {
```

```
    };

int main()
    {
    std::cout << "is_same<base, base> == " << std::boolalpha
        << std::is_same<base, base>::value << std::endl;
    std::cout << "is_same<base, derived> == " << std::boolalpha
        << std::is_same<base, derived>::value << std::endl;
    std::cout << "is_same<derived, base> == " << std::boolalpha
        << std::is_same<derived, base>::value << std::endl;
    std::cout << "is_same<int, int> == " << std::boolalpha
        << std::is_same<int, int>::value << std::endl;
    std::cout << "is_same<int, const int> == " << std::boolalpha
        << std::is_same<int, const int>::value << std::endl;

    return (0);
    }
```

Output

```
is_same<base, base> == true
is_same<base, derived> == false
is_same<derived, base> == false
is_same<int, int> == true
is_same<int, const int> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_convertible Class
is_base_of Class

is_scalar Class

Article • 08/03/2021

Tests if type is scalar.

Syntax

C++

```
template <class Ty>
struct is_scalar;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an integral type, a floating point type, an enumeration type, a pointer type, or a pointer to member type, or a `cv-qualified` form of one of them, otherwise it holds false.

Example

C++

```
// std__type_traits__is_scalar.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_scalar<trivial> == " << std::boolalpha
        << std::is_scalar<trivial>::value << std::endl;
    std::cout << "is_scalar<trivial *> == " << std::boolalpha
```

```
        << std::is_scalar<trivial *>::value << std::endl;
    std::cout << "is_scalar<int> == " << std::boolalpha
        << std::is_scalar<int>::value << std::endl;
    std::cout << "is_scalar<float> == " << std::boolalpha
        << std::is_scalar<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_scalar<trivial> == false
is_scalar<trivial *> == true
is_scalar<int> == true
is_scalar<float> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_compound Class

is_signed Class

Article • 08/03/2021

Test if type is signed integer.

Syntax

C++

```
template <class Ty>
struct is_signed;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a signed integral type or a `cv-qualified` signed integral type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_signed.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_signed<trivial> == " << std::boolalpha
        << std::is_signed<trivial>::value << std::endl;
    std::cout << "is_signed<int> == " << std::boolalpha
        << std::is_signed<int>::value << std::endl;
    std::cout << "is_signed<unsigned int> == " << std::boolalpha
```

```
        << std::is_signed<unsigned int>::value << std::endl;
    std::cout << "is_signed<float> == " << std::boolalpha
        << std::is_signed<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_signed<trivial> == false
is_signed<int> == true
is_signed<unsigned int> == false
is_signed<float> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_unsigned Class

is_standard_layout Class

Article • 08/03/2021

Tests if type is a standard layout.

Syntax

```
C++
```

```
template <class Ty>
struct is_standard_layout;
```

Parameters

Ty
The type to query

Remarks

An instance of this type predicate holds true if the type *Ty* is a class that has a standard layout of member objects in memory, otherwise it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivial Class

Article • 08/03/2021

Tests whether the type is a trivial type.

Syntax

```
C++
```

```
template <class T>
struct is_trivial;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a trivial type, otherwise it holds false. Trivial types are scalar types, trivially copyable class types, arrays of these types and cv-qualified versions of these types.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_assignable Class

Article • 08/03/2021

Tests whether a value of `From` type can be trivially assigned to `To` type

Syntax

C++

```
template <class To, class From>
struct is_trivially_assignable;
```

Parameters

To
The type of the object that receives the assignment.

From
The type of the object that provides the value.

Remarks

The expression `declval<To>() = declval<From>()` must be well-formed, and must be known to the compiler to require no non-trivial operations. Both `From` and `To` must be complete types, `void`, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_constructible Class

Article • 08/03/2021

Tests whether a type is trivially constructible when the specified argument types are used.

Syntax

```
C++
```

```
template <class T, class... Args>
struct is_trivially_constructible;
```

Parameters

T
The type to query.

Args
The argument types to match in a constructor of *T*.

Remarks

An instance of the type predicate holds true if the type *T* is trivially constructible by using the argument types in *Args*, otherwise it holds false. Type *T* is trivially constructible if the variable definition `T t(std::declval<Args>()...);` is well-formed and is known to call no non-trivial operations. Both *T* and all the types in *Args* must be complete types, `void`, or arrays of unknown bound.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_copy_assignable Class

Article • 08/03/2021

Tests whether the type has a trivial copy assignment operator.

Syntax

```
C++
```

```
template <class Ty>
struct is_trivially_copy_assignable;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class that has a trivial copy assignment operator, otherwise it holds false.

An assignment constructor for a class *T* is trivial if it is implicitly provided, the class *T* has no virtual functions, the class *T* has no virtual bases, the classes of all the non-static data members of class type have trivial assignment operators, and the classes of all the non-static data members of type array of class have trivial assignment operators.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_copy_constructible Class

Article • 08/03/2021

Tests if the type has a trivial copy constructor.

Syntax

```
C++
```

```
template <class T>
struct is_trivially_copy_constructible;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class that has a trivial copy constructor, otherwise it holds false.

A copy constructor for a class *T* is trivial if it is implicitly declared, the class *T* has no virtual functions or virtual bases, all the direct bases of class *T* have trivial copy constructors, the classes of all the non-static data members of class type have trivial copy constructors, and the classes of all the non-static data members of type array of class have trivial copy constructors.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_copyable Class

Article • 08/03/2021

Tests whether the type is a trivially copyable type.

Syntax

C++

```
template <class T>
struct is_trivially_copyable;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a trivially copyable type, otherwise it holds false. Trivially copyable types have no non-trivial copy operations, move operations, or destructors. Generally, a copy operation is considered trivial if it can be implemented as a bitwise copy. Both built-in types and arrays of trivially copyable types are trivially copyable.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_default_constructible Class

Article • 08/03/2021

Tests if type has trivial default constructor.

Syntax

```
C++
```

```
template <class Ty>
struct is_trivially_default_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial constructor, otherwise it holds false.

A default constructor for a class *Ty* is trivial if:

- it is an implicitly declared default constructor
- the class *Ty* has no virtual functions
- the class *Ty* has no virtual bases
- all the direct bases of the class *Ty* have trivial constructors
- the classes of all the non-static data members of class type have trivial constructors
- the classes of all the non-static data members of type array of class have trivial constructors

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_destructible Class

Article • 08/03/2021

Tests whether the type is trivially destructible.

Syntax

```
C++
```

```
template <class T>
struct is_trivially_destructible;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a destructible type, and the destructor is known to the compiler to use no non-trivial operations. Otherwise, it holds false.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_move_assignable Class

Article • 08/03/2021

Tests whether the type has a trivial move assignment operator.

Syntax

```
C++
```

```
template <class Ty>
struct is_trivially_move_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move assignment operator, otherwise it holds false.

A move assignment operator for a class *Ty* is trivial if:

- it is implicitly provided
- the class *Ty* has no virtual functions
- the class *Ty* has no virtual bases
- the classes of all the non-static data members of class type have trivial move assignment operators
- the classes of all the non-static data members of type array of class have trivial move assignment operators

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_trivially_move_constructible Class

Article • 08/03/2021

Tests if type has trivial move constructor.

Syntax

```
C++
```

```
template <class Ty>
struct is_trivially_move_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move constructor, otherwise it holds false.

A move constructor for a class *Ty* is trivial if:

it is implicitly declared

its parameter types are equivalent to those of an implicit declaration

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the class has no volatile non-static data members

all the direct bases of the class *Ty* have trivial move constructors

the classes of all the non-static data members of class type have trivial move constructors

the classes of all the non-static data members of type array of class have trivial move constructors

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_union Class

Article • 08/03/2021

Tests if type is a union.

Syntax

C++

```
template <class Ty>
struct is_union;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a union type or a `cv-qualified` form of a union type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_union.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

union ints
    {
    int in;
    long lo;
    };

int main()
```

```
    {
    std::cout << "is_union<trivial> == " << std::boolalpha
        << std::is_union<trivial>::value << std::endl;
    std::cout << "is_union<int> == " << std::boolalpha
        << std::is_union<int>::value << std::endl;
    std::cout << "is_union<ints> == " << std::boolalpha
        << std::is_union<ints>::value << std::endl;

    return (0);
    }
```

Output

```
is_union<trivial> == false
is_union<int> == false
is_union<ints> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_class Class

is_unsigned Class

Article • 08/03/2021

Tests if type is unsigned integer.

Syntax

C++

```
template <class Ty>
struct is_unsigned;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is an unsigned integral type or a `cv-qualified` unsigned integral type, otherwise it holds false.

Example

C++

```
// std__type_traits__is_unsigned.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_unsigned<trivial> == " << std::boolalpha
        << std::is_unsigned<trivial>::value << std::endl;
    std::cout << "is_unsigned<int> == " << std::boolalpha
        << std::is_unsigned<int>::value << std::endl;
    std::cout << "is_unsigned<unsigned int> == " << std::boolalpha
```

```
        << std::is_unsigned<unsigned int>::value << std::endl;
    std::cout << "is_unsigned<float> == " << std::boolalpha
        << std::is_unsigned<float>::value << std::endl;

    return (0);
    }
```

Output

```
is_unsigned<trivial> == false
is_unsigned<int> == false
is_unsigned<unsigned int> == true
is_unsigned<float> == false
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_signed Class

is_void Class

Article • 08/03/2021

Tests whether the type is void.

Syntax

C++

```
template <class T>
struct is_void;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is `void` or a cv-qualified form of `void`, otherwise it holds false.

Example

C++

```
// std__type_traits__is_void.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_void<trivial> == " << std::boolalpha
        << std::is_void<trivial>::value << std::endl;
    std::cout << "is_void<void()> == " << std::boolalpha
        << std::is_void<void()>::value << std::endl;
    std::cout << "is_void<void> == " << std::boolalpha
```

```
        << std::is_void<void>::value << std::endl;

    return (0);
    }
```

Output

```
is_void<trivial> == false
is_void<void()> == false
is_void<void> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

is_volatile Class

Article • 08/03/2021

Tests if type is volatile.

Syntax

C++

```
template <class Ty>
struct is_volatile;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if *Ty* is `volatile-qualified`.

Example

C++

```
// std__type_traits__is_volatile.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

struct trivial
    {
    int val;
    };

int main()
    {
    std::cout << "is_volatile<trivial> == " << std::boolalpha
        << std::is_volatile<trivial>::value << std::endl;
    std::cout << "is_volatile<volatile trivial> == " << std::boolalpha
        << std::is_volatile<volatile trivial>::value << std::endl;
    std::cout << "is_volatile<int> == " << std::boolalpha
        << std::is_volatile<int>::value << std::endl;
```

```
    std::cout << "is_volatile<volatile int> == " << std::boolalpha
        << std::is_volatile<volatile int>::value << std::endl;

    return (0);
    }
```

Output

```
is_volatile<trivial> == false
is_volatile<volatile trivial> == true
is_volatile<int> == false
is_volatile<volatile int> == true
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
is_const Class

make_signed Class

Article • 08/03/2021

Makes type or the smallest signed type greater than or equal in size to type.

Syntax

```
C++
```

```
template <class T>
struct make_signed;

template <class T>
using make_signed_t = typename make_signed<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if `is_signed<T>` holds true. Otherwise it is the smallest unsigned type `UT` for which `sizeof (T) <= sizeof (UT)`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

make_unsigned Class

Article • 08/03/2021

Makes type or the smallest unsigned type greater than or equal in size to type.

Syntax

```
C++
```

```
template <class T>
struct make_unsigned;

template <class T>
using make_unsigned_t = typename make_unsigned<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of the type modifier holds a modified-type that is *T* if `is_unsigned<T>` holds true. Otherwise it is the smallest signed type `ST` for which `sizeof (T) <= sizeof (ST)`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

rank Class

Article • 08/03/2021

Gets number of array dimensions.

Syntax

C++

```
template <class Ty>
struct rank;
```

Parameters

Ty
The type to query.

Remarks

The type query holds the value of the number of dimensions of the array type *Ty*, or 0 if *Ty* is not an array type.

Example

C++

```
// std__type_traits__rank.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main()
    {
    std::cout << "rank<int> == "
        << std::rank<int>::value << std::endl;
    std::cout << "rank<int[5]> == "
        << std::rank<int[5]>::value << std::endl;
    std::cout << "rank<int[5][10]> == "
        << std::rank<int[5][10]>::value << std::endl;

    return (0);
    }
```

Output

```
rank<int> == 0
rank<int[5]> == 1
rank<int[5][10]> == 2
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
extent Class

remove_all_extents Class

Article • 08/03/2021

Makes non array type from array type.

Syntax

C++

```
template <class T>
struct remove_all_extents;

template <class T>
using remove_all_extents_t = typename remove_all_extents<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_all_extents<T>` holds a modified-type that is the element type of the array type *T* with all array dimensions removed, or *T* if *T* is not an array type.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    std::cout << "remove_all_extents<int> == "
        << typeid(std::remove_all_extents_t<int>).name()
        << std::endl;
    std::cout << "remove_all_extents_t<int[5]> == "
        << typeid(std::remove_all_extents_t<int[5]>).name()
        << std::endl;
    std::cout << "remove_all_extents_t<int[5][10]> == "
        << typeid(std::remove_all_extents_t<int[5][10]>).name()
        << std::endl;
```

```
    return (0);
    }
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_extent Class

`remove_const` class

Article • 06/30/2022

Makes a non-`const` type from a type.

Syntax

C++

```
template <class T>
struct remove_const;

template <class T>
using remove_const_t = typename remove_const<T>::type;
```

Parameters

`T`

The type to modify.

Remarks

An instance of `remove_const<T>` holds a modified-type that is `T1` when *`T`* is of the form `const T1`, otherwise *`T`*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    int *p = (std::remove_const_t<const int>*)0;

    p = p;  // to quiet "unused" warning
    std::cout << "remove_const_t<const int> == "
        << typeid(*p).name() << std::endl;

    return (0);
    }
```

Output

```
remove_const_t<const int> == int
```

Requirements

Header: `<type_traits>`

Namespace: `std`

See also

<type_traits>
add_const class
remove_cv class

remove_cv Class

Article • 08/03/2021

Makes non const/volatile type from type.

Syntax

C++

```
template <class T>
struct remove_cv;

template <class T>
using remove_cv_t = typename remove_cv<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_cv<T>` holds a modified-type that is `T1` when *T* is of the form `const T1`, `volatile T1`, or `const volatile T1`, otherwise *T*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    int *p = (std::remove_cv_t<const volatile int> *)0;

    p = p;  // to quiet "unused" warning
    std::cout << "remove_cv_t<const volatile int> == "
        << typeid(*p).name() << std::endl;

    return (0);
    }
```

Output

```
remove_cv_t<const volatile int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_const Class
remove_volatile Class

remove_extent Class

Article • 08/03/2021

Makes element type from array type.

Syntax

C++

```
template <class T>
struct remove_extent;

template <class T>
using remove_extent_t = typename remove_extent<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_extent<T>` holds a modified-type that is `T1` when *T* is of the form `T1[N]`, otherwise *T*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    std::cout << "remove_extent_t<int> == "
        << typeid(std::remove_extent_t<int>).name()
        << std::endl;T
    std::cout << "remove_extent_t<int[5]> == "
        << typeid(std::remove_extent_t<int[5]>).name()
        << std::endl;T
    std::cout << "remove_extent_t<int[5][10]> == "
        << typeid(std::remove_extent_t<int[5][10]>).name()
        << std::endl;
```

```
    return (0);
    }
```

Output

```
remove_extent_t<int> == int
remove_extent_t<int[5]> == int
remove_extent_t<int[5][10]> == int [10]
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
remove_all_extents Class

remove_pointer Class

Article • 08/03/2021

Makes type from pointer to type.

Syntax

C++

```
template <class T>
struct remove_pointer;

template <class T>
using remove_pointer_t = typename remove_pointer<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_pointer<T>` holds a modified-type that is `T1` when *T* is of the form `T1*`, `T1* const`, `T1* volatile`, or `T1* const volatile`, otherwise *T*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    int *p = (std::remove_pointer_t<int *> *)0;

    p = p;  // to quiet "unused" warning
    std::cout << "remove_pointer_t<int *> == "
        << typeid(*p).name() << std::endl;

    return (0);
    }
```

Output

```
remove_pointer_t<int *> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
add_pointer Class

remove_reference Class

Article • 08/03/2021

Makes non reference type from type.

Syntax

C++

```
template <class T>
struct remove_reference;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_reference<T>` holds a modified-type that is `T1` when *T* is of the form `T1&`, otherwise *T*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    int *p = (std::remove_reference_t<int&> *)0;

    p = p;  // to quiet "unused" warning
    std::cout << "remove_reference_t<int&> == "
        << typeid(*p).name() << std::endl;

    return (0);
    }
```

Output

```
remove_reference_t<int&> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
add_lvalue_reference Class

remove_volatile Class

Article • 08/03/2021

Makes non volatile type from type.

Syntax

C++

```
template <class T>
struct remove_volatile;

template <class T>
using remove_volatile_t = typename remove_volatile<T>::type;
```

Parameters

T
The type to modify.

Remarks

An instance of `remove_volatile<T>` holds a modified-type that is `T1` when *T* is of the form `volatile T1`, otherwise *T*.

Example

C++

```
#include <type_traits>
#include <iostream>

int main()
    {
    int *p = (std::remove_volatile_t<volatile int> *)0;

    p = p;  // to quiet "unused" warning
    std::cout << " remove_volatile_t<volatile int> == "
        << typeid(*p).name() << std::endl;

    return (0);
    }
```

Output

```
remove_volatile_t<volatile int> == int
```

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
add_volatile Class

result_of Class

Article • 08/03/2021

Determines the return type of the callable type that takes the specified argument types. Added in C++14, deprecated in C++17.

Syntax

C++

```
template<class>
struct result_of; // Causes a static assert

template <class Fn, class... ArgTypes>
struct result_of<Fn(ArgTypes...)>;

// Helper type
template<class T>
   using result_of_t = typename result_of<T>::type;
```

Parameters

Fn
The callable type to query.

ArgTypes
The types of the argument list to the callable type to query.

Remarks

Use this template to determine at compile time the result type of `Fn(ArgTypes)`, where *Fn* is a callable type, reference to function, or reference to callable type, invoked using an argument list of the types in *ArgTypes*. The `type` member of the class template names the result type of `decltype(std::invoke(declval<Fn>(), declval<ArgTypes>()...))` if the unevaluated expression `std::invoke(declval<Fn>(), declval<ArgTypes>()...)` is well-formed. Otherwise, the class template has no member `type`. The type *Fn* and all types in the parameter pack *ArgTypes* must be complete types, `void`, or arrays of unknown bound. Deprecated in favor of invoke_result in C++17.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>
invoke_result class

underlying_type Class

Article • 08/03/2021

Produces the underlying integral type for an enumeration type.

Syntax

```
C++
```

```
template <class T>
struct underlying_type;
```

Parameters

T
The type to modify.

Remarks

The `type` member typedef of the class template names the underlying integral type of *T*, when *T* is an enumeration type, otherwise there is no member typedef `type`.

Requirements

Header: <type_traits>

Namespace: std

See also

<type_traits>

`<type_traits>` functions

Article • 12/07/2021

is_assignable
is_copy_assignable
is_copy_constructible
is_default_constructible
is_move_assignable
is_move_constructible
is_nothrow_move_assignable
is_nothrow_swappable
is_nothrow_swappable_with
is_swappable
is_swappable_with
is_trivially_copy_assignable
is_trivially_move_assignable
is_trivially_move_constructible

is_assignable

Tests whether a value of *From* type can be assigned to a *To* type.

```
C++
```

```
template <class To, class From>
struct is_assignable;
```

Parameters

To
The type of the object that receives the assignment.

From
The type of the object that provides the value.

Remarks

The unevaluated expression `declval<To>() = declval<From>()` must be well-formed.
Both *From* and *To* must be complete types, `void`, or arrays of unknown bound.

is_copy_assignable

Tests whether type has can be copied on assignment.

```
C++
```

```
template <class Ty>
struct is_copy_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy assignment operator, otherwise it holds false. Equivalent to is_assignable<Ty&, const Ty&>.

is_copy_constructible

Tests if type has a copy constructor.

```
C++
```

```
template <class Ty>
struct is_copy_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a copy constructor, otherwise it holds false.

Example

C++

```
#include <type_traits>
#include <iostream>

struct Copyable
{
    int val;
};

struct NotCopyable
{
   NotCopyable(const NotCopyable&) = delete;
   int val;

};

int main()
{
    std::cout << "is_copy_constructible<Copyable> == " << std::boolalpha
        << std::is_copy_constructible<Copyable>::value << std::endl;
    std::cout << "is_copy_constructible<NotCopyable> == " << std::boolalpha
        << std::is_copy_constructible<NotCopyable>::value << std::endl;

    return (0);
}
```

Output

```
is_copy_constructible<Copyable> == true
is_copy_constructible<NotCopyable > == false
```

is_default_constructible

Tests if a type has a default constructor.

C++

```
template <class Ty>
struct is_default_constructible;
```

Parameters

T

The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class type that has a default constructor, otherwise it holds false. This is equivalent to the predicate `is_constructible<T>`. Type *T* must be a complete type, **`void`**, or an array of unknown bound.

Example

C++

```
#include <type_traits>
#include <iostream>

struct Simple
{
    Simple() : val(0) {}
    int val;
};

struct Simple2
{
    Simple2(int v) : val(v) {}
    int val;
};

int main()
{
    std::cout << "is_default_constructible<Simple> == " << std::boolalpha
        << std::is_default_constructible<Simple>::value << std::endl;
    std::cout << "is_default_constructible<Simple2> == " << std::boolalpha
        << std::is_default_constructible<Simple2>::value << std::endl;

    return (0);
}
```

Output

```
is_default_constructible<Simple> == true
is_default_constructible<Simple2> == false
```

is_move_assignable

Tests if the type can be move assigned.

C++

```
template <class T>
struct is_move_assignable;
```

Parameters

T
The type to query.

Remarks

A type is move assignable if an rvalue reference to the type can be assigned to a reference to the type. The type predicate is equivalent to `is_assignable<T&, T&&>`. Move assignable types include referenceable scalar types and class types that have either compiler-generated or user-defined move assignment operators.

is_move_constructible

Tests whether the type has a move constructor.

C++

```
template <class T>
struct is_move_constructible;
```

Parameters

T
The type to be evaluated

Remarks

A type predicate that evaluates to true if the type *T* can be constructed by using a move operation. This predicate is equivalent to `is_constructible<T, T&&>`.

is_nothrow_move_assignable

Tests whether type has a **`nothrow`** move assignment operator.

C++

```
template <class Ty>
struct is_nothrow_move_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* has a nothrow move assignment operator, otherwise it holds false.

is_nothrow_swappable

C++

```
template <class T> struct is_nothrow_swappable;
```

is_nothrow_swappable_with

C++

```
template <class T, class U> struct is_nothrow_swappable_with;
```

is_swappable

C++

```
template <class T> struct is_swappable;
```

is_swappable_with

C++

```
template <class T, class U> struct is_swappable_with;
```

is_trivially_copy_assignable

Tests whether the type has a trivial copy assignment operator.

```
C++
```

```
template <class Ty>
struct is_trivially_copy_assignable;
```

Parameters

T
The type to query.

Remarks

An instance of the type predicate holds true if the type *T* is a class that has a trivial copy assignment operator, otherwise it holds false.

An assignment constructor for a class *T* is trivial if it is implicitly provided, the class *T* has no virtual functions, the class *T* has no virtual bases, the classes of all the non-static data members of class type have trivial assignment operators, and the classes of all the non-static data members of type array of class have trivial assignment operators.

is_trivially_move_assignable

Tests whether the type has a trivial move assignment operator.

```
C++
```

```
template <class Ty>
struct is_trivially_move_assignable;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move assignment operator, otherwise it holds false.

A move assignment operator for a class *Ty* is trivial if:

it is implicitly provided

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the classes of all the non-static data members of class type have trivial move assignment operators

the classes of all the non-static data members of type array of class have trivial move assignment operators

is_trivially_move_constructible

Tests if type has trivial move constructor.

```
C++
```

```
template <class Ty>
struct is_trivially_move_constructible;
```

Parameters

Ty
The type to query.

Remarks

An instance of the type predicate holds true if the type *Ty* is a class that has a trivial move constructor, otherwise it holds false.

A move constructor for a class *Ty* is trivial if:

it is implicitly declared

its parameter types are equivalent to those of an implicit declaration

the class *Ty* has no virtual functions

the class *Ty* has no virtual bases

the class has no volatile non-static data members

all the direct bases of the class *Ty* have trivial move constructors

the classes of all the non-static data members of class type have trivial move constructors

the classes of all the non-static data members of type array of class have trivial move constructors

See also

<type_traits>

`<type_traits>` typedefs

Article • 12/07/2021

false_type
true_type

false_type Typedef

Holds integral constant with false value.

C++

```
typedef integral_constant<bool, false> false_type;
```

Remarks

The type is a synonym for a specialization of the template `integral_constant`.

Example

C++

```
#include <type_traits>
#include <iostream>

int main() {
    std::cout << "false_type == " << std::boolalpha
        << std::false_type::value << std::endl;
    std::cout << "true_type == " << std::boolalpha
        << std::true_type::value << std::endl;

    return (0);
}
```

Output

```
false_type == false
true_type == true
```

true_type Typedef

Holds integral constant with true value.

C++

```
typedef integral_constant<bool, true> true_type;
```

Remarks

The type is a synonym for a specialization of the template `integral_constant`.

Example

C++

```
// std__type_traits__true_type.cpp
// compile with: /EHsc
#include <type_traits>
#include <iostream>

int main() {
    std::cout << "false_type == " << std::boolalpha
        << std::false_type::value << std::endl;
    std::cout << "true_type == " << std::boolalpha
        << std::true_type::value << std::endl;

    return (0);
}
```

Output

```
false_type == false
true_type == true
```

See also

<type_traits>

<typeindex>

Article • 12/07/2021

Include the standard header <typeindex> to define a class and function that support the indexing of objects of class type_info.

Syntax

```
C++

#include <typeindex>
```

Remarks

The hash Structure defines a `hash function` that's suitable for mapping values of type type_index to a distribution of index values.

The `type_index` class wraps a pointer to a `type_info` object to assist in indexing.

See also

Header Files Reference
Thread Safety in the C++ Standard Library
C++ Standard Library Reference

hash Structure

Article • 08/03/2021

The class template defines its method as returning `val.hash_code()`. The method defines a hash function that is used to map values of type type_index to a distribution of index values.

Syntax

C++

```
template <> struct hash<type_index>
: public unary_function<type_index, size_t>
{ // hashes a typeinfo object
    size_t operator()(type_index val) const;
};
```

Specialized Types

<system_error>

C++

```
template <class T> struct hash;
template <> struct hash<error_code>;
template <> struct hash<error_condition>;
```

See also

<typeindex>

type_index Class

Article • 06/21/2022

The `type_index` class wraps a pointer to type_info Class to assist in indexing by such objects.

class type_index { public: type_index(const type_info& tinfo); const char *name() const; size_t hash_code() const; bool operator==(const type_info& right) const; bool operator!=(const type_info& right) const; bool operator<(const type_info& right) const; bool operator<=(const type_info& right) const; bool operator>(const type_info& right) const; bool operator>=(const type_info& right) const; };

The constructor initializes `ptr` to `&tinfo`.

`name` returns `ptr->name()`.

`hash_code` returns `ptr->hash_code().`

`operator==` returns `*ptr == right.ptr`.

`operator!=` returns `!(*this == right)`.

`operator<` returns `*ptr->before(*right.ptr)`.

`operator<=` returns `!(right < *this).`

`operator>` returns `right < *this`.

`operator>=` returns `!(*this < right)`.

See also

Run-Time Type Information
<typeindex>

<typeinfo>

Article • 12/07/2021

Include the standard header <typeinfo> to define several types associated with the type-identification operator typeid Operator, which yields information about both static and dynamic types.

Syntax

```
C++

#include <typeinfo>
```

Remarks

For information on classes defined in <typeinfo>, see the following topics:

- bad_cast Exception
- bad_typeid Exception
- type_info Class

<unordered_map>

Article • 12/07/2021

Defines the container class templates unordered_map and unordered_multimap and their supporting templates.

Syntax

```
C++

#include <unordered_map>
```

> **Note**
>
> The <unordered_map> library also uses the `#include <initializer_list>` statement.

Classes

Class	Description
unordered_map Class	Stores hash table of {key, mapped} pairs.
unordered_multimap Class	Stores hash table of {key, mapped} pairs.

Functions

Function	Description
operator!=	Tests if the unordered_map object on the left side of the operator is not equal to the unordered_map object on the right side.
operator==	Tests if the unordered_map object on the left side of the operator is equal to the unordered_map object on the right side.
swap Function (unordered_map)	Swaps two maps.
operator!=	Tests if the unordered_multimap object on the left side of the operator is not equal to the unordered_multimap object on the right side.

Function	Description
operator==	Tests if the unordered_multimap object on the left side of the operator is equal to the unordered_multimap object on the right side.
swap Function (unordered_map)	Swaps two multimaps.

See also

unordered_multiset Class
unordered_set Class

`<unordered_map>` functions

Article • 12/07/2021

swap (unordered_map) swap (unordered_multimap)

swap (unordered_map)

Swaps the contents of two containers.

C++

```
template <class Key, class Ty, class Hash, class Pred, class Alloc>
void swap(
    unordered_map <Key, Ty, Hash, Pred, Alloc>& left,
    unordered_map <Key, Ty, Hash, Pred, Alloc>& right);
```

Parameters

Key
The key type.

Ty
The mapped type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc
The allocator class.

left
The first container to swap.

right
The second container to swap.

Remarks

The template function executes `left.`unordered_map::swap`(right)`.

Example

C++

```
// std__unordered_map__u_m_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

// display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    swap(c1, c2);

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
```

swap (unordered_multimap)

Swaps the contents of two containers.

C++

```
template <class Key, class Ty, class Hash, class Pred, class Alloc>
void swap(
    unordered_multimap <Key, Ty, Hash, Pred, Alloc>& left,
    unordered_multimap <Key, Ty, Hash, Pred, Alloc>& right);
```

Parameters

Key
The key type.

Ty
The mapped type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc
The allocator class.

left
The first container to swap.

right
The second container to swap.

Remarks

The template function executes `left.`unordered_multimap::swap`(right)`.

Example

C++

```
// std__unordered_map__u_mm_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

    // display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
```

See also

<unordered_map>

<unordered_map> operators

Article • 12/07/2021

unordered_map::operator!=
unordered_map::operator==
unordered_multimap::operator!=
unordered_multimap::operator==

operator!=

Tests whether the unordered_map object on the left side of the operator is not equal to the unordered_map object on the right side.

```
C++
```

```
bool operator!=(const unordered_map <Key, Type, Hash, Pred, Allocator>&
left, const unordered_map <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_map`.

right
An object of type `unordered_map`.

Return Value

`true` if the unordered_maps are not equal; `false` if they are equal.

Remarks

The comparison between unordered_map objects is not affected by the arbitrary order in which they store their elements. Two unordered_maps are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_map_op_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
   using namespace std;
   unordered_map<int, int> um1, um2, um3;

   for ( int i = 0 ; i < 3 ; ++i ) {
      um1.insert( make_pair( i+1, i ) );
      um1.insert( make_pair( i, i ) );

      um2.insert( make_pair( i, i+1 ) );
      um2.insert( make_pair( i, i ) );

      um3.insert( make_pair( i, i ) );
      um3.insert( make_pair( i+1, i ) );
   }

   cout << boolalpha;
   cout << "um1 != um2: " << (um1 != um2) << endl;
   cout << "um1 != um3: " << (um1 != um3) << endl;
   cout << "um2 != um3: " << (um2 != um3) << endl;
}

/* Output:
um1 != um2: true
um1 != um3: false
um2 != um3: true
*/
```

operator==

Tests whether the unordered_map object on the left side of the operator is equal to the unordered_map object on the right side.

C++

```
bool operator==(const unordered_map <Key, Type, Hash, Pred, Allocator>&
left, const unordered_map <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_map`.

right
An object of type `unordered_map`.

Return Value

`true` if the unordered_maps are equal; **`false`** if they are not equal.

Remarks

The comparison between unordered_map objects is not affected by the arbitrary order in which they store their elements. Two unordered_maps are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_map_op_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
   using namespace std;
   unordered_map<int, int> um1, um2, um3;

   for ( int i = 0 ; i < 3 ; ++i ) {
      um1.insert( make_pair( i+1, i ) );
      um1.insert( make_pair( i, i ) );

      um2.insert( make_pair( i, i+1 ) );
      um2.insert( make_pair( i, i ) );

      um3.insert( make_pair( i, i ) );
      um3.insert( make_pair( i+1, i ) );
   }

   cout << boolalpha;
   cout << "um1 == um2: " << (um1 == um2) << endl;
   cout << "um1 == um3: " << (um1 == um3) << endl;
   cout << "um2 == um3: " << (um2 == um3) << endl;
```

```
}

/* Output:
um1 == um2: false
um1 == um3: true
um2 == um3: false
*/
```

operator!= (multimap)

Tests whether the unordered_multimap object on the left side of the operator is not equal to the unordered_multimap object on the right side.

C++

```
bool operator!=(const unordered_multimap <Key, Type, Hash, Pred, Allocator>&
left, const unordered_multimap <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_multimap`.

right
An object of type `unordered_multimap`.

Return Value

`true` if the unordered_multimaps are not equal; **`false`** if they are equal.

Remarks

The comparison between unordered_multimap objects is not affected by the arbitrary order in which they store their elements. Two unordered_multimaps are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are not equal.

Example

C++

```
// unordered_multimap_op_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
   using namespace std;
   unordered_multimap<int, int> um1, um2, um3;

   for ( int i = 0 ; i < 3 ; ++i ) {
      um1.insert( make_pair( i, i ) );
      um1.insert( make_pair( i, i ) );

      um2.insert( make_pair( i, i ) );
      um2.insert( make_pair( i, i ) );
      um2.insert( make_pair( i, i ) );

      um3.insert( make_pair( i, i ) );
      um3.insert( make_pair( i, i ) );
   }

   cout << boolalpha;
   cout << "um1 != um2: " << (um1 != um2) << endl;
   cout << "um1 != um3: " << (um1 != um3) << endl;
   cout << "um2 != um3: " << (um2 != um3) << endl;
}

/* Output:
um1 != um2: true
um1 != um3: false
um2 != um3: true
*/
```

operator== (multimap)

Tests whether the unordered_multimap object on the left side of the operator is equal to the unordered_multimap object on the right side.

C++

```
bool operator==(const unordered_multimap <Key, Type, Hash, Pred, Allocator>&
left, const unordered_multimap <Key, Type, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_multimap`.

right
An object of type `unordered_multimap`.

Return Value

`true` if the unordered_multimaps are equal; **`false`** if they are not equal.

Remarks

The comparison between unordered_multimap objects is not affected by the arbitrary order in which they store their elements. Two unordered_multimaps are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_multimap_op_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_map>
#include <iostream>
#include <ios>

int main( )
{
   using namespace std;
   unordered_multimap<int, int> um1, um2, um3;

   for ( int i = 0 ; i < 3 ; ++i ) {
      um1.insert( make_pair( i, i ) );
      um1.insert( make_pair( i, i ) );

      um2.insert( make_pair( i, i ) );
      um2.insert( make_pair( i, i ) );
      um2.insert( make_pair( i, i ) );

      um3.insert( make_pair( i, i ) );
      um3.insert( make_pair( i, i ) );
   }

   cout << boolalpha;
   cout << "um1 == um2: " << (um1 == um2) << endl;
   cout << "um1 == um3: " << (um1 == um3) << endl;
```

```
    cout << "um2 == um3: " << (um2 == um3) << endl;
}

/* Output:
um1 == um2: false
um1 == um3: true
um2 == um3: false
*/
```

See also

<unordered_map>

`unordered_map` Class

Article • 06/21/2022

The class template describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket, a comparison function determines whether any pair of elements has equivalent ordering. Each element stores two objects, a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators that point at the removed element.

Syntax

C++

```
template <class Key,
    class Ty,
    class Hash = std::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<std::pair<const Key, Ty>>>
class unordered_map;
```

Parameters

`Key`

The key type.

`Ty`

The mapped type.

`Hash`

The hash function object type.

`Pred`

The equality comparison function object type.

`Alloc`

The allocator class.

Members

Type Definition	Description
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.
difference_type	The type of a signed distance between two elements.
hasher	The type of the hash function.
iterator	The type of an iterator for the controlled sequence.
key_equal	The type of the comparison function.
key_type	The type of an ordering key.
local_iterator	The type of a bucket iterator for the controlled sequence.
mapped_type	The type of a mapped value associated with each key.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.

Member Function	Description
at	Finds an element with the specified key.
begin	Designates the beginning of the controlled sequence.
bucket	Gets the bucket number for a key value.
bucket_count	Gets the number of buckets.
bucket_size	Gets the size of a bucket.

Member Function	Description
cbegin	Designates the beginning of the controlled sequence.
cend	Designates the end of the controlled sequence.
clear	Removes all elements.
count	Finds the number of elements matching a specified key.
contains C++20	Check if there's an element with the specified key in the `unordered_map`.
emplace	Adds an element constructed in place.
emplace_hint	Adds an element constructed in place, with hint.
empty	Tests whether no elements are present.
end	Designates the end of the controlled sequence.
equal_range	Finds range that matches a specified key.
erase	Removes elements at specified positions.
find	Finds an element that matches a specified key.
get_allocator	Gets the stored allocator object.
hash_function	Gets the stored hash function object.
insert	Adds elements.
key_eq	Gets the stored comparison function object.
load_factor	Counts the average elements per bucket.
max_bucket_count	Gets the maximum number of buckets.
max_load_factor	Gets or sets the maximum elements per bucket.
max_size	Gets the maximum size of the controlled sequence.
rehash	Rebuilds the hash table.
size	Counts the number of elements.
swap	Swaps the contents of two containers.
unordered_map	Constructs a container object.

Operator	Description
unordered_map::operator[]	Finds or inserts an element with the specified key.

Operator	Description
unordered_map::operator=	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type unordered_map::key_equal and a hash function object of type unordered_map::hasher. You access the first stored object by calling the member function unordered_map::key_eq`()`; and you access the second stored object by calling the member function unordered_map::hash_function`()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike class template unordered_multimap Class, an object of type `unordered_map` ensures that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys are unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes unordered_map::load_factor`()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You can't in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type unordered_map::allocator_type. Such an allocator object must have the same external interface as an object of type `allocator`. The stored allocator object isn't copied when the container object is assigned.

Requirements

Header: `<unordered_map>`

Namespace: std

`unordered_map::allocator_type`

The type of an allocator for managing storage.

C++

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

C++

```
// std__unordered_map__unordered_map_allocator_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

Output

```
al == std::allocator() is true
```

`unordered_map::at`

Finds an element in a unordered_map with a specified key value.

C++

```
Ty& at(const Key& key);
const Ty& at(const Key& key) const;
```

Parameters

key

The key value to find.

Return Value

A reference to the data value of the element found.

Remarks

If the argument key value isn't found, then the function throws an object of class `out_of_range`.

Example

C++

```
// unordered_map_at.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // find and show elements
    std::cout << "c1.at('a') == " << c1.at('a') << std::endl;
    std::cout << "c1.at('b') == " << c1.at('b') << std::endl;
    std::cout << "c1.at('c') == " << c1.at('c') << std::endl;

    return (0);
}
```

`unordered_map::begin`

Designates the beginning of the controlled sequence or a bucket.

C++

```
iterator begin();
const_iterator begin() const;
local_iterator begin(size_type nbucket);
const_local_iterator begin(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

C++

```
// std__unordered_map__unordered_map_begin.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // inspect first two items " [c 3] [b 2]"
    Mymap::iterator it2 = c1.begin();
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    ++it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    std::cout << std::endl;
```

```
    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2]
[a, 1]
```

`unordered_map::bucket`

Gets the bucket number for a key value.

C++

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

C++

```
// std__unordered_map__unordered_map_bucket.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
```

```
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_map::bucket_count

Gets the number of buckets.

C++

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

C++

```
// std__unordered_map__unordered_map_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
```

```
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3][b, 2][a, 1]
bucket_count() == 8
```

```
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_map::bucket_size

Gets the size of a bucket

C++

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The member functions return the size of bucket number *nbucket*.

Example

C++

```
// std__unordered_map__unordered_map_bucket_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;
```

```
    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

`unordered_map::cbegin`

Returns a **`const`** iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A **`const`** forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++
```

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

`unordered_map::cend`

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
C++
```

```
const_iterator cend() const;
```

Return Value

A `const` forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();
// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`unordered_map::clear`

Removes all elements.

C++

```
void clear();
```

Remarks

The member function calls `unordered_map::erase(unordered_map::begin(), unordered_map::end())`, see unordered_map::erase, unordered_map::begin, and unordered_map::end.

Example

C++

```
// std__unordered_map__unordered_map_clear.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;
```

```
    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

`unordered_map::const_iterator`

The type of a constant iterator for the controlled sequence.

C++

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It's described here as a synonym for the implementation-defined type `T1`.

Example

C++

```
// std__unordered_map__unordered_map_const_iterator.cpp
// compile with: /EHsc
```

```
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

C++

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It's described here as a synonym for the implementation-defined type `T5`.

Example

C++

```
// std__unordered_map__unordered_map_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
```

```
typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

`unordered_map::const_pointer`

The type of a constant pointer to an element.

C++

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_const_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
    {
        Mymap::const_pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::const_reference

The type of a constant reference to an element.

C++

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_const_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
    {
        Mymap::const_reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
```

`unordered_map::contains`

Checks if there's an element in the `unordered_map` with the specified key. Introduced in C++20.

C++

```
bool contains(const Key& key) const;
<class K> bool contains(const K& key) const;
```

Parameters

K

The type of the key.

key

The key value of the element to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <unordered_map>
#include <iostream>

int main()
{
    std::unordered_map<int, bool> theUnorderedMap = {{0, false}, {1,true}};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theUnorderedMap.contains(1) << '\n';
    std::cout << theUnorderedMap.contains(2) << '\n';

    return 0;
}
```

Output

```
true
false
```

`unordered_map::count`

Finds the number of elements matching a specified key.

C++

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by unordered_map::equal_range(keyval).

Example

C++

```
// std__unordered_map__unordered_map_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
count('A') == 0
count('b') == 1
count('C') == 0
```

unordered_map::difference_type

The type of a signed distance between two elements.

C++

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It's described here as a synonym for the implementation-defined type `T3`.

Example

C++

```
// std__unordered_map__unordered_map_difference_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // compute positive difference
    Mymap::difference_type diff = 0;
    for (Mymap::const_iterator it = c1.begin();
```

```
        it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Mymap::const_iterator it = c1.end();
        it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
end()-begin() == 3
begin()-end() == -3
```

`unordered_map::emplace`

Inserts an element constructed in place (no copy or move operations are performed) into an unordered_map.

C++

```
template <class... Args>
pair<iterator, bool>  emplace( Args&&... args);
```

Parameters

`args`

The arguments forwarded to construct an element to be inserted into the `unordered_map` unless it already contains an element whose value is equivalently ordered.

Return Value

A `pair` whose **`bool`** component returns true if an insertion was made and false if the `unordered_map` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **`bool`** component of a pair `pr` returned by this member function, use `pr.second`.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but doesn't occur in the container's hash function, the container isn't modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see map::emplace.

`unordered_map::emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
C++
```

```
template <class... Args>
iterator emplace_hint(const_iterator where, Args&&... args);
```

Parameters

`args`
The arguments forwarded to construct an element to be inserted into the unordered_map unless the unordered_map already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.

`where`
A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No references are invalidated by this function.

During the insertion, if an exception is thrown but doesn't occur in the container's hash function, the container isn't modified. If the exception is thrown in the hash function, the result is undefined.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see map::emplace_hint.

`unordered_map::empty`

Tests whether no elements are present.

C++

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_empty.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
```

```
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

    // display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_map::end

Designates the end of the controlled sequence.

C++

```
iterator end();
const_iterator end() const;
local_iterator end(size_type nbucket);
const_local_iterator end(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

unordered_map::equal_range

Finds range that matches a specified key.

```
C++
```

```
std::pair<iterator, iterator>  equal_range(const Key& keyval);
std::pair<const_iterator, const_iterator>  equal_range(const Key& keyval)
const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

```
C++
```

```
// std__unordered_map__unordered_map_equal_range.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
```

```
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // display results of failed search
    std::pair<Mymap::iterator, Mymap::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
        << ", " << pair1.first->second << "]";
    std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
        << ", " << pair1.first->second << "]";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
equal_range('x'):
equal_range('b'): [b, 2]
```

unordered_map::erase

Removes an element or a range of elements in a unordered_map from specified positions or removes elements that match a specified key.

C++

```
iterator erase(const_iterator Where);
iterator erase(const_iterator First, const_iterator Last);
size_type erase(const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last

Position just beyond the last element to be removed.

Key

The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the unordered_map.

Remarks

For a code example, see map::erase.

unordered_map::find

Finds an element that matches a specified key.

```
C++
```

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns unordered_map::equal_range(keyval).first.

Example

C++

```
// std__unordered_map__unordered_map_find.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Mymap::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << it->first << ", " << it->second << "]" << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
find('A') == false
find('b') == true: [b, 2]
```

unordered_map::get_allocator

Gets the stored allocator object.

C++

```
Alloc get_allocator() const;
```

Remarks

The member function returns the stored allocator object.

Example

C++

```
// std__unordered_map__unordered_map_get_allocator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

Output

```
al == std::allocator() is true
```

unordered_map::hash_function

Gets the stored hash function object.

C++

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

C++

```
// std__unordered_map__unordered_map_hash_function.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

`unordered_map::hasher`

The type of the hash function.

C++

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

C++

```cpp
// std__unordered_map__unordered_map_hasher.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_map::insert

Inserts an element or a range of elements into an unordered_map.

C++

```cpp
// (1) single element
pair<iterator, bool> insert(    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(    ValTy&& Val);

// (3) single element with hint
iterator insert(    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(InputIterator First,
    InputIterator Last);
```

```
// (6) initializer list
void insert(initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to be inserted into the unordered_map unless it already contains an element whose key is equivalently ordered.

Where

The place to start searching for the correct point of insertion.

ValTy

Template parameter that specifies the argument type that the unordered_map can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First

The position of the first element to be copied.

Last

The position just beyond the last element to be copied.

InputIterator

Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

IList

The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose **bool** component is `true` if an insertion was made, and `false` if the `unordered_map` already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is `true`, or to the existing element if the **bool** component is `false`.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the unordered_map or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown but doesn't occur in the container's hash function, the container's state isn't modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair` `pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and for map, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into an unordered_map that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` doesn't get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `m`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the unordered_map.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see unordered_map::emplace and unordered_map::emplace_hint.

For a code example, see map::insert.

`unordered_map::iterator`

The type of an iterator for the controlled sequence.

```
C++

typedef T0 iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for the controlled sequence. It's described here as a synonym for the implementation-defined type `T0`.

Example

C++

```
// std__unordered_map__unordered_map_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

`unordered_map::key_eq`

Gets the stored comparison function object.

C++

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

C++

```
// std__unordered_map__unordered_map_key_eq.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
    }
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

`unordered_map::key_equal`

The type of the comparison function.

C++

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

C++

```
// std__unordered_map__unordered_map_key_equal.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
    }
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

`unordered_map::key_type`

The type of an ordering key.

C++

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

C++

```cpp
// std__unordered_map__unordered_map_key_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

unordered_map::load_factor

Counts the average elements per bucket.

C++

```cpp
float load_factor() const;
```

Remarks

The member function returns `(float)unordered_map::size() / (float)unordered_map::bucket_count()`, the average number of elements per bucket, see unordered_map::size and unordered_map::bucket_count.

Example

C++

```
// std__unordered_map__unordered_map_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
```

```
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`unordered_map::local_iterator`

The type of a bucket iterator.

C++

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It's described here as a synonym for the implementation-defined type `T4`.

Example

C++

```
// std__unordered_map__unordered_map_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect bucket containing 'a'
    Mymap::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

`unordered_map::mapped_type`

The type of a mapped value associated with each key.

C++

```
typedef Ty mapped_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

C++

```
// std__unordered_map__unordered_map_mapped_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

unordered_map::max_bucket_count

Gets the maximum number of buckets.

C++

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

C++

```
// std__unordered_map__unordered_map_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
```

```
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`unordered_map::max_load_factor`

Gets or sets the maximum elements per bucket.

C++

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

`factor`

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *`factor`*.

Example

C++

```
// std__unordered_map__unordered_map_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;
```

```
    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`unordered_map::max_size`

Gets the maximum size of the controlled sequence.

C++

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

C++

```
// std__unordered_map__unordered_map_max_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
```

```
    Mymap c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
    }
```

Output

```
max_size() == 536870911
```

`unordered_map::operator[]`

Finds or inserts an element with the specified key.

C++

```
Ty& operator[](const Key& keyval);

Ty& operator[](Key&& keyval);
```

Parameters

`keyval`
The key value to find or insert.

Return Value

A reference to the data value of the inserted element.

Remarks

If the argument key value isn't found, then it's inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map `m` using *`m[Key] = DataValue;`* where *`DataValue`* is the value of the `mapped_type` of the element with a key value of *`Key`*.

The member function determines the iterator `where` as the return value of `unordered_map::insert(unordered_map::value_type(keyval, Ty())`. For more information, see unordered_map::insert and unordered_map::value_type. (It inserts an element with

the specified key if no such element exists.) It then returns a reference to `(*where).second`.

When using `operator[]` to insert elements, the returned reference doesn't indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions find and insert can be used to determine whether an element with a specified key is already present before an insertion.

Example

C++

```
// std__unordered_map__unordered_map_operator_sub.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
#include <string>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// try to find and fail
    std::cout << "c1['A'] == " << c1['A'] << std::endl;

// try to find and succeed
    std::cout << "c1['a'] == " << c1['a'] << std::endl;

// redisplay contents
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// insert by moving key
    std::unordered_map<string, int> c2;
    std::string str("abc");
    std::cout << "c2[std::move(str)] == " << c2[std::move(str)] <<
std::endl;
    std::cout << "c2["abc"] == " << c2["abc"] << std::endl;
```

```
    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
c1['A'] == 0
c1['a'] == 1
[c, 3] [b, 2] [A, 0] [a, 1]
c2[move(str)] == 0
c2["abc"] == 1
```

`unordered_map::operator=`

Replaces the elements of this unordered_map using the elements from another unordered_map.

C++

```
unordered_map& operator=(const unordered_map& right);

unordered_map& operator=(unordered_map&& right);
```

Parameters

`right`
The unordered_map that the operator function assigns content from.

Remarks

The first version copies all of the elements from *`right`* to this unordered_map.

The second version moves all of the elements from *`right`* to this unordered_map.

Any elements that are in this unordered_map before `operator=` executes are discarded.

Example

C++

```
// unordered_map_operator_as.cpp
// compile with: /EHsc
```

```
#include <unordered_map>
#include <iostream>

int main( )
   {
   using namespace std;
   unordered_map<int, int> v1, v2, v3;
   unordered_map<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
   }
```

`unordered_map::pointer`

The type of a pointer to an element.

C++

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

C++

```cpp
// std__unordered_map__unordered_map_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

`unordered_map::reference`

The type of a reference to an element.

C++

```cpp
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_map::rehash

Rebuilds the hash table.

C++

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

C++

```
// std__unordered_map__unordered_map_rehash.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1
```

`unordered_map::size`

Counts the number of elements.

C++

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
```

```
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

// display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

`unordered_map::size_type`

The type of an unsigned distance between two elements.

C++

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It's described here as a synonym for the implementation-defined

type `T2`.

Example

C++

```
// std__unordered_map__unordered_map_size_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;
    Mymap::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
    }
```

Output

```
size == 0
```

`unordered_map::swap`

Swaps the contents of two containers.

C++

```
void swap(unordered_map& right);
```

Parameters

`right`

The container to swap with.

Remarks

The member function swaps the controlled sequences between **`*this`** and *`right`*. If `unordered_map::get_allocator() == right.get_allocator()`, see

unordered_map::get_allocator, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

C++

```
// std__unordered_map__unordered_map_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

// display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    swap(c1, c2);

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;
```

```
    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
```

unordered_map::unordered_map

Constructs a container object.

C++

```
unordered_map(const unordered_map& Right);

explicit unordered_map(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Allocator());

unordered_map(unordered_map&& Right);
unordered_map(initializer_list<Type> IList);
unordered_map(initializer_list<Type> IList, size_type Bucket_count);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    KeyEqual& equal);

unordered_map(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    KeyEqual& Equal
    const Allocator& Al);

template <class InIt>
unordered_map(
    InputIterator First,
    InputIterator Last,
```

```
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());
```

Parameters

Al

The allocator object to store.

Comp

The comparison function object to store.

Hash

The hash function object to store.

Bucket_count

The minimum number of buckets.

Right

The container to copy.

First

The position of the first element to be copied.

Last

The position just beyond the last element to be copied.

IList

The initializer_list that contains the elements to be copied.

Remarks

The first constructor specifies a copy of the sequence controlled by `right`. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[first, last)`. The fourth constructor specifies a copy of the sequence by moving `right`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *`Right`*. Otherwise:

the minimum number of buckets is the argument *`Bucket_count`*, if present; otherwise it's a default value described here as the implementation-defined value `N0`.

the hash function object is the argument *Hash*, if present; otherwise it's `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it's `Pred()`.

The allocator object is the argument *Al*, if present; otherwise, it's `Alloc()`.

Example

C++

```
// std__unordered_map__unordered_map_construct.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
#include <initializer_list>

using namespace std;

using Mymap = unordered_map<char, int>;

int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    Mymap c2(8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    // display contents " [f 6] [e 5] [d 4]"
    for (const auto& c : c2) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    Mymap c3(c1.begin(),
        c1.end(),
        8,
```

```
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c3) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    Mymap c4(move(c3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c4) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;
    cout << endl;

    // Construct with an initializer_list
    unordered_map<int, char> c5({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8,
'j' } });
    for (const auto& c : c5) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size
    unordered_map<int, char> c6({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8,
'j' } }, 4);
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;
    cout << endl;

    // Initializer_list plus size and hash
    unordered_map<int, char, hash<char>> c7(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>()
    );

    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size, hash, and key_equal
    unordered_map<int, char, hash<char>, equal_to<char>> c8(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>()
    );
```

```
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size, hash, key_equal, and allocator
    unordered_map<int, char, hash<char>, equal_to<char>> c9(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >()
    );

    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;
}
```

Output

```
[a, 1] [b, 2] [c, 3]
[d, 4] [e, 5] [f, 6]
[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]

[5, g] [6, h] [7, i] [8, j]
[a, 1] [b, 2] [c, 3]

[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3]
```

`unordered_map::value_type`

The type of an element.

C++

```
typedef std::pair<const Key, Ty> value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_map_value_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_map<char, int> Mymap;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    // add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

See also

<unordered_map>
Thread Safety in the C++ Standard Library\

unordered_multimap Class

Article • 08/17/2021

The class template describes an object that controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element stores two objects, a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

C++

```
template <class Key,
    class Ty,
    class Hash = std::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key>>
class unordered_multimap;
```

Parameters

Key
The key type.

Ty
The mapped type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc

The allocator class.

Members

Type Definition	Description
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.
difference_type	The type of a signed distance between two elements.
hasher	The type of the hash function.
iterator	The type of an iterator for the controlled sequence.
key_equal	The type of the comparison function.
key_type	The type of an ordering key.
local_iterator	The type of a bucket iterator for the controlled sequence.
mapped_type	The type of a mapped value associated with each key.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.

Member Function	Description
begin	Designates the beginning of the controlled sequence.
bucket	Gets the bucket number for a key value.
bucket_count	Gets the number of buckets.
bucket_size	Gets the size of a bucket.
cbegin	Designates the beginning of the controlled sequence.

Member Function	Description
cend	Designates the end of the controlled sequence.
clear	Removes all elements.
contains C++20	Checks if there is an element with the specified key in the `unordered_multimap`.
count	Finds the number of elements matching a specified key.
emplace	Adds an element constructed in place.
emplace_hint	Adds an element constructed in place, with hint.
empty	Tests whether no elements are present.
end	Designates the end of the controlled sequence.
equal_range	Finds range that matches a specified key.
erase	Removes elements at specified positions.
find	Finds an element that matches a specified key.
get_allocator	Gets the stored allocator object.
hash_function	Gets the stored hash function object.
insert	Adds elements.
key_eq	Gets the stored comparison function object.
load_factor	Counts the average elements per bucket.
max_bucket_count	Gets the maximum number of buckets.
max_load_factor	Gets or sets the maximum elements per bucket.
max_size	Gets the maximum size of the controlled sequence.
rehash	Rebuilds the hash table.
size	Counts the number of elements.
swap	Swaps the contents of two containers.
unordered_multimap	Constructs a container object.

Operator	Description
unordered_multimap::operator=	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type unordered_multimap::key_equal and a hash function object of type unordered_multimap::hasher. You access the first stored object by calling the member function unordered_multimap::key_eq `()`; and you access the second stored object by calling the member function unordered_multimap::hash_function `()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike class template unordered_map Class, an object of type `unordered_multimap` does not ensure that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys need not be unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes unordered_multimap::load_factor `()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type unordered_multimap::allocator_type. Such an allocator object must have the same external interface as an object of type `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: <unordered_map>

Namespace: std

unordered_multimap::allocator_type

The type of an allocator for managing storage.

```
C++
```

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

C++

```
// std__unordered_map__unordered_multimap_allocator_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
    {
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
    }
```

Output

```
al == std::allocator() is true
```

unordered_multimap::begin

Designates the beginning of the controlled sequence or a bucket.

C++

```
iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

C++

```
// std__unordered_map__unordered_multimap_begin.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect first two items " [c 3] [b 2]"
    Mymap::iterator it2 = c1.begin();
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    ++it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    std::cout << std::endl;

// inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";
```

```
    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2]
[a, 1]
```

unordered_multimap::bucket

Gets the bucket number for a key value.

C++

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval
The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

C++

```
// std__unordered_map__unordered_multimap_bucket.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));
```

```
// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multimap::bucket_count

Gets the number of buckets.

C++

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

C++

```
// std__unordered_map__unordered_multimap_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
```

```
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
```

```
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multimap::bucket_size

Gets the size of a bucket

C++

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

C++

```
// std__unordered_map__unordered_multimap_bucket_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
```

```
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// display buckets for keys
    Mymap::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multimap::cbegin

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```C++
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

unordered_multimap::cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```C++
const_iterator cend() const;
```

Return Value

A `const` forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```C++
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

unordered_multimap::clear

Removes all elements.

```
C++
```

```
void clear();
```

Remarks

The member function calls unordered_multimap::erase(unordered_multimap::begin(), unordered_multimap::end()).

Example

```
C++
```

```
// std__unordered_map__unordered_multimap_clear.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

// display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
```

```
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::const_iterator

The type of a constant iterator for the controlled sequence.

C++

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

C++

```
// std__unordered_map__unordered_multimap_const_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
```

```
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

C++

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

C++

```
// std__unordered_map__unordered_multimap_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;
```

```
    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

unordered_multimap::const_pointer

The type of a constant pointer to an element.

C++

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_const_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
```

```
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::const_pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::const_reference

The type of a constant reference to an element.

C++

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_const_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
```

```
typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::const_reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::contains

Checks if there is an element with the specified key in the `unordered_multimap`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

K
The type of the key.

key
The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <unordered_map>
#include <iostream>

int main()
{
    std::unordered_multimap<int, bool> theUnorderedMultimap = {{0, false},
{1,true}};

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theUnorderedMultimap.contains(1) << '\n';
    std::cout << theUnorderedMultimap.contains(2) << '\n';

    return 0;
}
```

Output

```
true
false
```

unordered_multimap::count

Finds the number of elements matching a specified key.

C++

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by unordered_multimap::equal_range`(keyval)`.

Example

C++

```
// std__unordered_map__unordered_multimap_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
count('A') == 0
count('b') == 1
count('C') == 0
```

unordered_multimap::difference_type

The type of a signed distance between two elements.

C++

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

C++

```
// std__unordered_map__unordered_multimap_difference_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// compute positive difference
    Mymap::difference_type diff = 0;
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

// compute negative difference
    diff = 0;
    for (Mymap::const_iterator it = c1.end();
        it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;
```

```
    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
end()-begin() == 3
begin()-end() == -3
```

unordered_multimap::emplace

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace(Args&&... args);
```

Parameters

args
The arguments forwarded to construct an element to be inserted into the `unordered_multimap`.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see multimap::emplace.

unordered_multimap::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
C++
```

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

args
The arguments forwarded to construct an element to be inserted into the unordered.

where
A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

The value_type of an element is a pair, so that the value of an element will be an ordered pair with the first component equal to the key value and the second component equal to the data value of the element.

For a code example, see map::emplace_hint.

unordered_multimap::empty

Tests whether no elements are present.

C++

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_empty.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

// display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
```

```
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::end

Designates the end of the controlled sequence.

C++

```
iterator end();

const_iterator end() const;

local_iterator end(size_type nbucket);

const_local_iterator end(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

C++

```
// std__unordered_map__unordered_multimap_end.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect last two items " [a 1] [b 2]"
    Mymap::iterator it2 = c1.end();
    --it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    --it2;
    std::cout << " [" << it2->first << ", " << it2->second << "]";
    std::cout << std::endl;

// inspect bucket containing 'a'
    Mymap::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[a, 1] [b, 2]
[a, 1]
```

unordered_multimap::equal_range

Finds range that matches a specified key.

C++

```
std::pair<iterator, iterator>
    equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
    equal_range(const Key& keyval) const;
```

Parameters

keyval
Key value to search for.

Remarks

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

C++

```
// std__unordered_map__unordered_multimap_equal_range.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// display results of failed search
    std::pair<Mymap::iterator, Mymap::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
```

```
            << ", " << pair1.first->second << "]";
    std::cout << std::endl;

// display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << " [" << pair1.first->first
            << ", " << pair1.first->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
equal_range('x'):
equal_range('b'): [b, 2]
```

unordered_multimap::erase

Removes an element or a range of elements in a unordered_multimap from specified positions or removes elements that match a specified key.

C++

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where

Position of the element to be removed.

First

Position of the first element to be removed.

Last
Position just beyond the last element to be removed.

Key
The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the map if no such element exists.

For the third member function, returns the number of elements that have been removed from the unordered_multimap.

Remarks

For a code example, see map::erase.

unordered_multimap::find

Finds an element that matches a specified key.

```
C++
```

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval
Key value to search for.

Remarks

The member function returns unordered_multimap::equal_range`(keyval).first`.

Example

```
C++
```

```
// std__unordered_map__unordered_multimap_find.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

// try to find and succeed
    Mymap::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << it->first << ", " << it->second << "]" << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
find('A') == false
find('b') == true: [b, 2]
```

unordered_multimap::get_allocator

Gets the stored allocator object.

C++

```
Alloc get_allocator() const;
```

Remarks

The member function returns the stored allocator object.

Example

C++

```
// std__unordered_map__unordered_multimap_get_allocator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
    {
    Mymap c1;

    Mymap::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
    }
```

Output

```
al == std::allocator() is true
```

unordered_multimap::hash_function

Gets the stored hash function object.

C++

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

C++

```
// std__unordered_map__unordered_multimap_hash_function.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
    }
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multimap::hasher

The type of the hash function.

C++

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

C++

```
// std__unordered_map__unordered_multimap_hasher.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>
```

```
typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    Mymap::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
    }
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multimap::insert

Inserts an element or a range of elements into an unordered_multimap.

C++

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);
```

```
// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val
The value of an element to be inserted into the unordered_multimap.

Where
The place to start searching for the correct point of insertion.

ValTy
Template parameter that specifies the argument type that the unordered_multimap can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First
The position of the first element to be copied.

Last
The position just beyond the last element to be copied.

InputIterator
Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

IList
The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the unordered_multimap.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the unordered_multimap.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The value_type of a container is a typedef that belongs to the container, and for map, `map<K, V>::value_type` is `pair<const K, V>`. The value of an element is an ordered pair in which the first component is equal to the key value and the second component is equal to the data value of the element.

The range member function (5) inserts the sequence of element values into an unordered_multimap that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *Last* does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an initializer_list to copy elements into the unordered_multimap.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see unordered_multimap::emplace and unordered_multimap::emplace_hint.

For a code example, see multimap::insert.

unordered_multimap::iterator

The type of an iterator for the controlled sequence.

```
C++
```

```
typedef T0 iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T0`.

Example

```
C++
```

```
// std__unordered_map__unordered_multimap_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::key_eq

Gets the stored comparison function object.

C++

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

C++

```
// std__unordered_map__unordered_multimap_key_eq.cpp
// compile with: /EHsc
```

```
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
    }
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multimap::key_equal

The type of the comparison function.

C++

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

C++

```
// std__unordered_map__unordered_multimap_key_equal.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
```

```
    Mymap c1;

    Mymap::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
    }
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multimap::key_type

The type of an ordering key.

C++

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

C++

```
// std__unordered_map__unordered_multimap_key_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));
```

```
// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

unordered_multimap::load_factor

Counts the average elements per bucket.

C++

```
float load_factor() const;
```

Remarks

The member function returns `(float)` unordered_multimap::size `() /` `(float)` unordered_multimap::bucket_count `()`, the average number of elements per bucket.

Example

C++

```
// std__unordered_map__unordered_multimap_load_factor.cpp
// compile with: /EHsc
```

```
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

unordered_multimap::local_iterator

The type of a bucket iterator.

C++

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

C++

```
// std__unordered_map__unordered_multimap_local_iterator.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect bucket containing 'a'
    Mymap::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << " [" << lit->first << ", " << lit->second << "]";

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[a, 1]
```

unordered_multimap::mapped_type

The type of a mapped value associated with each key.

```
C++
```

```
typedef Ty mapped_type;
```

Remarks

The type is a synonym for the template parameter `Ty`.

Example

```
C++
```

```
// std__unordered_map__unordered_multimap_mapped_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

unordered_multimap::max_bucket_count

Gets the maximum number of buckets.

C++

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

C++

```
// std__unordered_map__unordered_multimap_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
```

```
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multimap::max_load_factor

Gets or sets the maximum elements per bucket.

C++

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor
The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example

C++

```
// std__unordered_map__unordered_multimap_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;
```

```
// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multimap::max_size

Gets the maximum size of the controlled sequence.

C++

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

C++

```
// std__unordered_map__unordered_multimap_max_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
    }
```

Output

```
max_size() == 536870911
```

unordered_multimap::operator=

Copies a hash table.

C++

```
unordered_multimap& operator=(const unordered_multimap& right);

unordered_multimap& operator=(unordered_multimap&& right);
```

Parameters

right
The `unordered_multimap` being copied into the `unordered_multimap`.

Remarks

After erasing any existing elements in a unordered_multimap, `operator=` either copies or moves the contents of *right* into the unordered_multimap.

Example

C++

```
// unordered_multimap_operator_as.cpp
// compile with: /EHsc
#include <unordered_multimap>
#include <iostream>

int main( )
   {
   using namespace std;
   unordered_multimap<int, int> v1, v2, v3;
   unordered_multimap<int, int>::iterator iter;

   v1.insert(pair<int, int>(1, 10));

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << iter->second << " ";
   cout << endl;
   }
```

unordered_multimap::pointer

The type of a pointer to an element.

C++

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_pointer.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::pointer p = &*it;
        std::cout << " [" << p->first << ", " << p->second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::reference

The type of a reference to an element.

C++

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_reference.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Mymap::reference ref = *it;
        std::cout << " [" << ref.first << ", " << ref.second << "]";
        }
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::rehash

Rebuilds the hash table.

C++

```
void rehash(size_type nbuckets);
```

Parameters

nbuckets
The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

C++

```
// std__unordered_map__unordered_multimap_rehash.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

// change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
```

```
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

// rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1
```

unordered_multimap::size

Counts the number of elements.

C++

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

C++

```cpp
// std__unordered_map__unordered_multimap_size.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert(Mymap::value_type('d', 4));
    c1.insert(Mymap::value_type('e', 5));

// display contents " [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
size == 0
empty() == true

[e, 5] [d, 4]
size == 2
empty() == false
```

unordered_multimap::size_type

The type of an unsigned distance between two elements.

C++

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

C++

```
// std__unordered_map__unordered_multimap_size_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;
    Mymap::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
    }
```

Output

```
size == 0
```

unordered_multimap::swap

Swaps the contents of two containers.

C++

```
void swap(unordered_multimap& right);
```

Parameters

right
The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If unordered_multimap::get_allocator `() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

C++

```
// std__unordered_map__unordered_multimap_swap.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    Mymap c2;

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
```

```
    c2.insert(Mymap::value_type('f', 6));

    c1.swap(c2);

// display contents " [f 6] [e 5] [d 4]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    swap(c1, c2);

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::unordered_multimap

Constructs a container object.

C++

```
unordered_multimap(
    const unordered_multimap& Right);

explicit unordered_multimap(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Pred(),
    const Allocator& Al = Alloc());

unordered_multimap(
    unordered_multimap&& Right);

unordered_multimap(
    initializer_list<Type> IList);

unordered_multimap(
    initializer_list<Type> IList,
```

```
    size_type Bucket_count);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key);

unordered_multimap(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key,
    const Allocator& Al);

template <class InputIterator>
unordered_multimap(
    InputIterator first,
    InputIterator last,
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Pred(),
    const Allocator& Al = Alloc());
```

Parameters

InputIterator

The iterator type.

Al

The allocator object to store.

Comp

The comparison function object to store.

Hash

The hash function object to store.

Bucket_count

The minimum number of buckets.

Right

The container to copy.

IList
The initializer_list from which to copy the elements.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor specifies an empty controlled sequence. The third constructor. specifies a copy of the sequence by moving *Right*. The fourth, fifth, sixth, seventh, and eighth constructors use an initializer_list for the members. The ninth constructor inserts the sequence of element values `[First, Last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *Bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `N0`.

The hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Pred()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

Example

C++

```
// std__unordered_map__unordered_multimap_construct.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

using namespace std;

using  Mymap = unordered_multimap<char, int> ;
int main()
{
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
```

```
    cout << endl;

    Mymap c2(8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    c2.insert(Mymap::value_type('d', 4));
    c2.insert(Mymap::value_type('e', 5));
    c2.insert(Mymap::value_type('f', 6));

    // display contents " [f 6] [e 5] [d 4]"
    for (const auto& c : c2) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    Mymap c3(c1.begin(),
        c1.end(),
        8,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >());

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c3) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    Mymap c4(move(c3));

    // display contents " [c 3] [b 2] [a 1]"
    for (const auto& c : c4) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Construct with an initializer_list
    unordered_multimap<int, char> c5({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, {
8, 'j' } });
    for (const auto& c : c5) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size
    unordered_multimap<int, char> c6({ { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, {
8, 'j' } }, 4);
    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size and hash
```

```
    unordered_multimap<int, char, hash<char>> c7(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>()
    );

    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size, hash, and key_equal
    unordered_multimap<int, char, hash<char>, equal_to<char>> c8(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>()
    );

    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;

    // Initializer_list plus size, hash, key_equal, and allocator
    unordered_multimap<int, char, hash<char>, equal_to<char>> c9(
        { { 5, 'g' }, { 6, 'h' }, { 7, 'i' }, { 8, 'j' } },
        4,
        hash<char>(),
        equal_to<char>(),
        allocator<pair<const char, int> >()
    );

    for (const auto& c : c1) {
        cout << " [" << c.first << ", " << c.second << "]";
    }
    cout << endl;
}
```

Output

```
[a, 1] [b, 2] [c, 3] [d, 4] [e, 5] [f, 6] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2]
[c, 3] [5, g] [6, h] [7, i] [8, j] [a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3]
[a, 1] [b, 2] [c, 3] [a, 1] [b, 2] [c, 3] [c, 3] [b, 2] [a, 1]
[f, 6] [e, 5] [d, 4]
[c, 3] [b, 2] [a, 1]
[c, 3] [b, 2] [a, 1]
```

unordered_multimap::value_type

The type of an element.

C++

```
typedef std::pair<const Key, Ty> value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

C++

```
// std__unordered_map__unordered_multimap_value_type.cpp
// compile with: /EHsc
#include <unordered_map>
#include <iostream>

typedef std::unordered_multimap<char, int> Mymap;
int main()
    {
    Mymap c1;

    c1.insert(Mymap::value_type('a', 1));
    c1.insert(Mymap::value_type('b', 2));
    c1.insert(Mymap::value_type('c', 3));

// display contents " [c 3] [b 2] [a 1]"
    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

// add a value and reinspect
    Mymap::key_type key = 'd';
    Mymap::mapped_type mapped = 4;
    Mymap::value_type val = Mymap::value_type(key, mapped);
    c1.insert(val);

    for (Mymap::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << it->first << ", " << it->second << "]";
    std::cout << std::endl;

    return (0);
    }
```

Output

```
[c, 3] [b, 2] [a, 1]
[d, 4] [c, 3] [b, 2] [a, 1]
```

See also

<unordered_map>

Containers

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

<unordered_set>

Article • 12/07/2021

Defines the container class templates unordered_multiset and unordered_set and their supporting templates.

Requirements

Header: <unordered_set>

Namespace: std

> **Note**
>
> The <unordered_set> library also uses the `#include <initializer_list>` statement.

Members

Classes

Name	Description
unordered_multiset Class	Stores hash table of keys.
unordered_set Class	Stores hash table of keys.

Functions

Name	Description
operator!=	Tests if the unordered_multiset object on the left side of the operator is not equal to the unordered_multiset object on the right side.
operator==	Tests if the unordered_multiset object on the left side of the operator is equal to the unordered_multiset object on the right side.
swap	Swaps two multisets.
operator!=	Tests if the unordered_set object on the left side of the operator is not equal to the unordered_set object on the right side.

Name	Description
operator==	Tests if the unordered_set object on the left side of the operator is equal to the unordered_set object on the right side.
swap	Swaps two sets.

See also

unordered_map Class
unordered_multimap Class

`<unordered_set>` functions

Article • 12/07/2021

swap (unordered_set)

Swaps the contents of two containers.

C++

```
template <class Key, class Hash, class Pred, class Alloc>
void swap(
   unordered_set <Key, Hash, Pred, Alloc>& left,
   unordered_set <Key, Hash, Pred, Alloc>& right);
```

Parameters

Key
The key type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc
The allocator class.

left
The first container to swap.

right
The second container to swap.

Remarks

The template function executes `left.`unordered_set::swap`(right)`.

Example

C++

```
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
Myset c1;

c1.insert('a');
c1.insert('b');
c1.insert('c');

// display contents " [c] [b] [a]"
for (Myset::const_iterator it = c1.begin();
it != c1.end(); ++it)
std::cout << " [" << *it << "]";
std::cout << std::endl;

Myset c2;

c2.insert('d');
c2.insert('e');
c2.insert('f');

c1.swap(c2);

// display contents " [f] [e] [d]"
for (Myset::const_iterator it = c1.begin();
it != c1.end(); ++it)
std::cout << " [" << *it << "]";
std::cout << std::endl;

swap(c1, c2);

// display contents " [c] [b] [a]"
for (Myset::const_iterator it = c1.begin();
it != c1.end(); ++it)
std::cout << " [" << *it << "]";
std::cout << std::endl;

return (0);
}
```

Output

```
[c] [b] [a]
[f] [e] [d]
[c] [b] [a]
```

swap (unordered_multiset)

Swaps the contents of two containers.

```
C++
```

```
template <class Key, class Hash, class Pred, class Alloc>
void swap(
   unordered_multiset <Key, Hash, Pred, Alloc>& left,
   unordered_multiset <Key, Hash, Pred, Alloc>& right);
```

Parameters

Key
The key type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc
The allocator class.

left
The first container to swap.

right
The second container to swap.

Remarks

The template function executes `left.`unordered_multiset::swap`(right)`.

Example

```
C++
```

```
// std__unordered_set__u_ms_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
```

```
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << *it << "]";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents " [f] [e] [d]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << *it << "]";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents " [c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << " [" << *it << "]";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[f] [e] [d]
[c] [b] [a]
```

`<unordered_set>` operators

Article • 12/07/2021

operator!=

Tests whether the unordered_set object on the left side of the operator is not equal to the unordered_set object on the right side.

C++

```
bool operator!=(const unordered_set <Key, Hash, Pred, Allocator>& left,
const unordered_set <Key, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_set`.

right
An object of type `unordered_set`.

Return Value

`true` if the unordered_sets are not equal; **`false`** if they are equal.

Remarks

The comparison between unordered_set objects is not affected by the arbitrary order in which they store their elements. Two unordered_sets are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_set_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
```

```
#include <ios>

int main()
{
    using namespace std;

    unordered_set<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

   cout << boolalpha;
   cout << "c1 != c2: " << (c1 != c2) << endl;
   cout << "c1 != c3: " << (c1 != c3) << endl;
   cout << "c2 != c3: " << (c2 != c3) << endl;

    return (0);
}
```

Output:

```
c1 != c2: true
```

```
c1 != c3: false
```

```
c2 != c3: true
```

operator==

Tests whether the unordered_set object on the left side of the operator is equal to the unordered_set object on the right side.

```
C++
```

```
bool operator==(const unordered_set <Key, Hash, Pred, Allocator>& left,
const unordered_set <Key, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_set`.

right
An object of type `unordered_set`.

Return Value

`true` if the unordered_sets are equal; **`false`** if they are not equal.

Remarks

The comparison between unordered_set objects is not affected by the arbitrary order in which they store their elements. Two unordered_sets are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_set_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_set<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

   cout << boolalpha;
   cout << "c1 == c2: " << (c1 == c2) << endl;
```

```
    cout << "c1 == c3: " << (c1 == c3) << endl;
    cout << "c2 == c3: " << (c2 == c3) << endl;

     return (0);
}
```

Output

```
c1 == c2: false
c1 == c3: true
c2 == c3: false
```

operator!= (multiset)

Tests whether the unordered_multiset object on the left side of the operator is not equal to the unordered_multiset object on the right side.

C++

```
bool operator!=(const unordered_multiset <Key, Hash, Pred, Allocator>& left,
const unordered_multiset <Key, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_multiset`.

right
An object of type `unordered_multiset`.

Return Value

`true` if the unordered_multisets are not equal; **`false`** if they are equal.

Remarks

The comparison between unordered_multiset objects is not affected by the arbitrary order in which they store their elements. Two unordered_multisets are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_multiset_ne.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_multiset<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
    c1.insert('c');

    c2.insert('c');
    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

   cout << boolalpha;
   cout << "c1 != c2: " << (c1 != c2) << endl;
   cout << "c1 != c3: " << (c1 != c3) << endl;
   cout << "c2 != c3: " << (c2 != c3) << endl;

    return (0);
}
```

Output

```
c1 != c2: true
c1 != c3: false
c2 != c3: true
```

operator== (multiset)

Tests whether the unordered_multiset object on the left side of the operator is equal to the unordered_multiset object on the right side.

C++

```
bool operator==(const unordered_multiset <Key, Hash, Pred, Allocator>& left,
const unordered_multiset <Key, Hash, Pred, Allocator>& right);
```

Parameters

left
An object of type `unordered_multiset`.

right
An object of type `unordered_multiset`.

Return Value

`true` if the unordered_multisets are equal; **`false`** if they are not equal.

Remarks

The comparison between unordered_multiset objects is not affected by the arbitrary order in which they store their elements. Two unordered_multisets are equal if they have the same number of elements and the elements in one container are a permutation of the elements in the other container. Otherwise, they are unequal.

Example

C++

```
// unordered_multiset_eq.cpp
// compile by using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>
#include <ios>

int main()
{
    using namespace std;

    unordered_multiset<char> c1, c2, c3;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
    c1.insert('c');
```

```
    c2.insert('c');
    c2.insert('c');
    c2.insert('a');
    c2.insert('d');

    c3.insert('c');
    c3.insert('c');
    c3.insert('a');
    c3.insert('b');

   cout << boolalpha;
   cout << "c1 == c2: " << (c1 == c2) << endl;
   cout << "c1 == c3: " << (c1 == c3) << endl;
   cout << "c2 == c3: " << (c2 == c3) << endl;

    return (0);
}
```

Output

```
c1 == c2: false
c1 == c3: true
c2 == c3: false
```

`unordered_set` Class

Article • 02/21/2022

The class template describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket, a comparison function determines whether any pair of elements has equivalent ordering. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Inserting an element invalidates no iterators, and removing an element invalidates only those iterators that point at the removed element.

Syntax

C++

```
template <
   class Key,
   class Hash = std::hash<Key>,
   class Pred = std::equal_to<Key>,
   class Alloc = std::allocator<Key>>
class unordered_set;
```

Parameters

`Key`

The key type.

`Hash`

The hash function object type.

`Pred`

The equality comparison function object type.

`Alloc`

The allocator class.

Members

Typedefs

Name	Description
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.
difference_type	The type of a signed distance between two elements.
hasher	The type of the hash function.
iterator	The type of an iterator for the controlled sequence.
key_equal	The type of the comparison function.
key_type	The type of an ordering key.
local_iterator	The type of a bucket iterator for the controlled sequence.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.

Functions

Name	Description
begin	Designates the beginning of the controlled sequence.
bucket	Gets the bucket number for a key value.
bucket_count	Gets the number of buckets.
bucket_size	Gets the size of a bucket.
cbegin	Designates the beginning of the controlled sequence.

Name	Description
cend	Designates the end of the controlled sequence.
clear	Removes all elements.
contains C++20	Check if there's an element with the specified key in the `unordered_set`.
count	Finds the number of elements matching a specified key.
emplace	Adds an element constructed in place.
emplace_hint	Adds an element constructed in place, with hint.
empty	Tests whether no elements are present.
end	Designates the end of the controlled sequence.
equal_range	Finds range that matches a specified key.
erase	Removes elements at specified positions.
find	Finds an element that matches a specified key.
get_allocator	Gets the stored allocator object.
hash_function	Gets the stored hash function object.
insert	Adds elements.
key_eq	Gets the stored comparison function object.
load_factor	Counts the average elements per bucket.
max_bucket_count	Gets the maximum number of buckets.
max_load_factor	Gets or sets the maximum elements per bucket.
max_size	Gets the maximum size of the controlled sequence.
rehash	Rebuilds the hash table.
size	Counts the number of elements.
swap	Swaps the contents of two containers.
unordered_set	Constructs a container object.

Operators

Name	Description

Name	Description
unordered_set::operator=	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type unordered_set::key_equal and a hash function object of type unordered_set::hasher. You access the first stored object by calling the member function unordered_set::key_eq `()`; and you access the second stored object by calling the member function unordered_set::hash_function `()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike class template unordered_multiset Class, an object of type `unordered_set` ensures that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys are unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes unordered_set::load_factor `()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You can't in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type unordered_set::allocator_type. Such an allocator object must have the same external interface as an object of type `allocator`. The stored allocator object isn't copied when the container object is assigned.

`unordered_set::allocator_type`

The type of an allocator for managing storage.

```
C++
```

```
typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

C++

```
// std__unordered_set__unordered_set_allocator_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
    << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

Output

```
al == std::allocator() is true
```

`begin`

Designates the beginning of the controlled sequence or a bucket.

C++

```
iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;
```

Parameters

nbucket

The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

C++

```
// unordered_set_begin.cpp
// compile using: cl.exe /EHsc /nologo /W4 /MTd
#include <unordered_set>
#include <iostream>

using namespace std;

typedef unordered_set<char> MySet;

int main()
{
    MySet c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents using range-based for
    for (auto it : c1) {
    cout << "[" << it << "] ";
    }

    cout << endl;

    // display contents using explicit for
    for (MySet::const_iterator it = c1.begin(); it != c1.end(); ++it) {
        cout << "[" << *it << "] ";
    }

    cout << std::endl;

    // display first two items
    MySet::iterator it2 = c1.begin();
    cout << "[" << *it2 << "] ";
    ++it2;
    cout << "[" << *it2 << "] ";
```

```
    cout << endl;

    // display bucket containing 'a'
    MySet::const_local_iterator lit = c1.begin(c1.bucket('a'));
    cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[a] [b] [c]
[a] [b] [c]
[a] [b]
[a]
```

bucket

Gets the bucket number for a key value.

C++

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval

The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value *keyval*.

Example

C++

```
// std__unordered_set__unordered_set_bucket.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
```

```
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
    << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

bucket_count

Gets the number of buckets.

C++

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

C++

```
// std__unordered_set__unordered_set_bucket_count.cpp
// compile with: /EHsc
```

```
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
```

```
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`bucket_size`

Gets the size of a bucket

C++

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

`nbucket`

The bucket number.

Remarks

The member functions returns the size of bucket number *`nbucket`*.

Example

C++

```
// std__unordered_set__unordered_set_bucket_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;
```

```
    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
    << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

cbegin

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with

the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container of any kind that supports `begin()` and `cbegin()`.

```
C++

auto i1 = Container.begin();
// i1 isContainer<T>::iterator
auto i2 = Container.cbegin();

// i2 isContainer<T>::const_iterator
```

cend

Returns a **`const`** iterator that addresses the location just beyond the last element in a range.

```
C++

const_iterator cend() const;
```

Return Value

A **`const`** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **`const`**) container of any kind that supports `end()` and `cend()`.

```
C++

auto i1 = Container.end();
// i1 isContainer<T>::iterator
auto i2 = Container.cend();

// i2 isContainer<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced.

`clear`

Removes all elements.

```
C++
```

```
void clear();
```

Remarks

The member function calls `unordered_set::erase( unordered_set::begin(), unordered_set::end())`. For more information, see unordered_set::erase, unordered_set::begin, and unordered_set::end.

Example

```
C++
```

```
// std__unordered_set__unordered_set_clear.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');
```

```
    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true
[e] [d]
size == 2
empty() == false
```

`const_iterator`

The type of a constant iterator for the controlled sequence.

C++

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It's described here as a synonym for the implementation-defined type `T1`.

Example

C++

```
// std__unordered_set__unordered_set_const_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
```

```
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
    std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

`const_local_iterator`

The type of a constant bucket iterator for the controlled sequence.

C++

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It's described here as a synonym for the implementation-defined type `T5`.

Example

C++

```
// std__unordered_set__unordered_set_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;
```

```
    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a]
```

`const_pointer`

The type of a constant pointer to an element.

C++

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_const_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
```

```
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::const_pointer p = &*it;
        std::cout << "[" << *p << "] ";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

const_reference

The type of a constant reference to an element.

C++

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_const_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
```

```
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::const_reference ref = *it;
        std::cout << "[" << ref << "] ";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

`contains`

Checks if there's an element with the specified key in the `unordered_set`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

`K`

The type of the key.

`Key`

The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <unordered_set>
#include <iostream>

int main()
{
    std::unordered_set<int> theUnorderedSet = { 1, 2 };

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theUnorderedSet.contains(2) << '\n';
    std::cout << theUnorderedSet.contains(3) << '\n';

    return 0;
}
```

Output

```
true
false
```

count

Finds the number of elements matching a specified key.

C++

```
size_type count(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by unordered_set::equal_range(keyval).

Example

C++

```
// std__unordered_set__unordered_set_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
count('A') == 0
count('b') == 1
count('C') == 0
```

`difference_type`

The type of a signed distance between two elements.

C++

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It's described here as a synonym for the implementation-defined type `T3`.

Example

C++

```
// std__unordered_set__unordered_set_difference_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // compute positive difference
    Myset::difference_type diff = 0;
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Myset::const_iterator it = c1.end(); it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
end()-begin() == 3
begin()-end() == -3
```

`emplace`

Inserts an element constructed in place (no copy or move operations are performed).

```
C++
```

```
template <class... Args>
pair<iterator, bool>
emplace(
Args&&... args);
```

Parameters

`args`
The arguments forwarded to construct an element to be inserted into the `unordered_set` unless it already contains an element whose value is equivalently ordered.

Return Value

A `pair` whose **`bool`** component returns true if an insertion was made and false if the `unordered_set` already contained an element whose key had an equivalent value in the ordering, and whose iterator component returns the address where a new element was inserted or where the element was already located.

To access the iterator component of a pair `pr` returned by this member function, use `pr.first`, and to dereference it, use `*(pr.first)`. To access the **`bool`** component of a pair `pr` returned by this member function, use `pr.second`.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but doesn't occur in the container's hash function, the container isn't modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see set::emplace.

`emplace_hint`

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
C++
```

```
template <class... Args>
iterator emplace_hint(
const_iteratorwhere,
Args&&... args);
```

Parameters

args

The arguments forwarded to construct an element to be inserted into the `unordered_set` unless the `unordered_set` already contains that element or, more generally, unless it already contains an element whose key is equivalently ordered.

where

A hint about the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No iterators or references are invalidated by this function.

During the insertion, if an exception is thrown but doesn't occur in the container's hash function, the container isn't modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see set::emplace_hint.

`empty`

Tests whether no elements are present.

```
C++
```

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_empty.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true
[e] [d]
size == 2
empty() == false
```

end

Designates the end of the controlled sequence.

C++

```
iterator end();

const_iterator end() const;

local_iterator end(size_type nbucket);

const_local_iterator end(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

C++

```
// std__unordered_set__unordered_set_end.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
```

```
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect last two items "[a] [b] "
    Myset::iterator it2 = c1.end();
    --it2;
    std::cout << "[" << *it2 << "] ";
    --it2;
    std::cout << "[" << *it2 << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a] [b]
[a]
```

equal_range

Finds range that matches a specified key.

C++

```
std::pair<iterator, iterator>
equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
equal_range(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

C++

```
// std__unordered_set__unordered_set_equal_range.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // display results of failed search
    std::pair<Myset::iterator, Myset::iterator> pair1 =
    c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "] ";
    std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
equal_range('x'):
equal_range('b'): [b]
```

erase

Removes an element or a range of elements in a `unordered_set` from specified positions or removes elements that match a specified key.

C++

```
iterator erase(const_iterator Where);

iterator erase(const_iterator First, const_iterator Last);

size_type erase(const key_type& Key);
```

Parameters

`Where`
Position of the element to be removed.

`First`
Position of the first element to be removed.

`Last`
Position just beyond the last element to be removed.

`Key`
The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the `unordered_set` if no such element exists.

For the third member function, returns the number of elements that have been removed from the `unordered_set`.

Remarks

For a code example, see set::erase.

`find`

Finds an element that matches a specified key.

C++

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval

Key value to search for.

Remarks

The member function returns unordered_set::equal_range`(keyval).first`.

Example

C++

```
// std__unordered_set__unordered_set_find.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // try to find and fail
```

```
    std::cout << "find('A') == "
    << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;

    // try to find and succeed
    Myset::iterator it = c1.find('b');
    std::cout << "find('b') == "
    << std::boolalpha << (it != c1.end())
    << ": [" << *it << "] " << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
find('A') == false
find('b') == true: [b]
```

`get_allocator`

Gets the stored allocator object.

C++

```
Alloc get_allocator() const;
```

Remarks

The member function returns the stored allocator object.

Example

C++

```
// std__unordered_set__unordered_set_get_allocator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
```

```
    std::cout << "al == std::allocator() is "
    << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

Output

```
al == std::allocator() is true
```

hash_function

Gets the stored hash function object.

C++

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

C++

```
// std__unordered_set__unordered_set_hash_function.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

hasher

The type of the hash function.

C++

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

C++

```
// std__unordered_set__unordered_set_hasher.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

insert

Inserts an element or a range of elements into an `unordered_set`.

```
C++
```

```
// (1) single element
pair<iterator, bool> insert(const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool> insert(ValTy&& Val);

// (3) single element with hint
iterator insert(const_iterator Where, const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(const_iterator Where, ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(InputIterator First, InputIterator Last);

// (6) initializer list
void insert(initializer_list<value_type> IList);
```

Parameters

`Val`

The value of an element to be inserted into the `unordered_set` unless it already contains an element whose key is equivalently ordered.

`Where`

The place to start searching for the correct point of insertion.

`ValTy`

Template parameter that specifies the argument type that the `unordered_set` can use to construct an element of value_type, and perfect-forwards *`Val`* as an argument.

`First`

The position of the first element to be copied.

`Last`

The position just beyond the last element to be copied.

`InputIterator`

Template function argument that meets the requirements of an input iterator that points

to elements of a type that can be used to construct value_type objects.

IList
The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose **bool** component is true if an insertion was made, and false if the unordered_set already contained an element whose key had an equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the **bool** component is true, or to the existing element if the **bool** component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the unordered_set or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown but doesn't occur in the container's hash function, the container's state isn't modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a pair pr that's returned by the single-element member functions, use pr.first; to dereference the iterator within the returned pair, use *pr.first, giving you an element. To access the **bool** component, use pr.second. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and, for set, unordered_set<V>::value_type is type const V.

The range member function (5) inserts the sequence of element values into an unordered_set that corresponds to each element addressed by an iterator in the range [First, Last); therefore, *Last* doesn't get inserted. The container member function end() refers to the position just after the last element in the container—for example, the statement s.insert(v.begin(), v.end()); attempts to insert all elements of v into s. Only elements that have unique values in the range are inserted; duplicates are

ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the `unordered_set`.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see set::emplace and set::emplace_hint.

For a code example, see set::insert.

`iterator`

A type that provides a constant forward iterator that can read elements in an unordered_set.

```
C++
```

```
typedef implementation-defined iterator;
```

Example

See the example for begin for an example of how to declare and use an **iterator**.

`key_eq`

Gets the stored comparison function object.

```
C++
```

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```
C++
```

```cpp
// std__unordered_set__unordered_set_key_eq.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
    << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
    << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

`key_equal`

The type of the comparison function.

C++

```cpp
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

C++

```cpp
// std__unordered_set__unordered_set_key_equal.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>
```

```
typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
    << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
    << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

key_type

The type of an ordering key.

C++

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

C++

```
// std__unordered_set__unordered_set_key_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
```

```
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[d] [c] [b] [a]
```

`load_factor`

Counts the average elements per bucket.

C++

```
float load_factor() const;
```

Remarks

The member function returns `(float)` unordered_set::size `() /` `(float)` unordered_set::bucket_count `()`, the average number of elements per bucket.

Example

C++

```
// std__unordered_set__unordered_set_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
```

```
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
```

```
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`local_iterator`

The type of a bucket iterator.

C++

```
typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It's described here as a synonym for the implementation-defined type `T4`.

Example

C++

```
// std__unordered_set__unordered_set_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
```

```
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a]
```

max_bucket_count

Gets the maximum number of buckets.

C++

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

C++

```
// std__unordered_set__unordered_set_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
```

```
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`max_load_factor`

Gets or sets the maximum elements per bucket.

```
C++
```

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

`factor`

The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *`factor`*.

Example

```
C++
```

```
// std__unordered_set__unordered_set_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
```

```
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
    << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
    << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

`max_size`

Gets the maximum size of the controlled sequence.

C++

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

C++

```
// std__unordered_set__unordered_set_max_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}
```

Output

```
max_size() == 4294967295
```

`operator=`

Copies a hash table.

C++

```
unordered_set& operator=(const unordered_set& right);

unordered_set& operator=(unordered_set&& right);
```

Parameters

right

The unordered_set being copied into the `unordered_set`.

Remarks

After erasing any existing elements in an `unordered_set`, `operator=` either copies or moves the contents of *right* into the `unordered_set`.

Example

C++

```
// unordered_set_operator_as.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

int main( )
{
    using namespace std;
    unordered_set<int> v1, v2, v3;
    unordered_set<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

`pointer`

The type of a pointer to an element.

C++

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::pointer p = &key;
        std::cout << "[" << *p << "] ";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

`reference`

The type of a reference to an element.

C++

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::iterator it = c1.begin(); it != c1.end(); ++it)
    {
        Myset::key_type key = *it;
        Myset::reference ref = key;
        std::cout << "[" << ref << "] ";
    }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

`rehash`

Rebuilds the hash table.

```
C++
```

```
void rehash(size_type nbuckets);
```

Parameters

`nbuckets`

The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *`nbuckets`* and rebuilds the hash table as needed.

Example

```
C++
```

```
// std__unordered_set__unordered_set_rehash.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;
```

```
    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1
```

`size`

Counts the number of elements.

C++

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

C++

```cpp
// std__unordered_set__unordered_set_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

`size_type`

The type of an unsigned distance between two elements.

```
C++
```

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It's described here as a synonym for the implementation-defined type `T2`.

Example

```
C++
```

```
// std__unordered_set__unordered_set_size_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;
    Myset::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

```
Output
```

```
size == 0
```

`swap`

Swaps the contents of two containers.

C++

```
void swap(unordered_set& right);
```

Parameters

right

The container to swap with.

Remarks

The member function swaps the controlled sequences between ***this** and *right*. If unordered_set::get_allocator() == right.get_allocator(), it does so in constant time, it throws an exception only as a result of copying the stored traits object of type Tr, and it invalidates no references, pointers, or iterators that designate elements in the two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

C++

```
// std__unordered_set__unordered_set_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
```

```
    c2.insert('f');

    c1.swap(c2);

    // display contents "[f] [e] [d] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[f] [e] [d]
[c] [b] [a]
```

unordered_set

Constructs a container object.

C++

```
unordered_set(const unordered_set& Right);

explicit unordered_set(
    size_typebucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

unordered_set(unordered_set&& Right);

unordered_set(initializer_list<Type> IList);

unordered_set(initializer_list<Type> IList, size_typebucket_count);

unordered_set(
    initializer_list<Type> IList,
    size_typebucket_count,
    const Hash& Hash);
```

```
unordered_set(
    initializer_list<Type> IList,
    size_typebucket_count,
    const Hash& Hash,
    const Comp& Comp);

unordered_set(
    initializer_list<Type> IList,
    size_typebucket_count,
    const Hash& Hash,
    const Comp& Comp,
    const Allocator& Al);

template <class InputIterator>
unordered_set(
    InputIteratorfirst,
    InputIteratorlast,
    size_typebucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());
```

Parameters

InputIterator

The iterator type.

Al

The allocator object to store.

Comp

The comparison function object to store.

Hash

The hash function object to store.

bucket_count

The minimum number of buckets.

Right

The container to copy.

IList

The `initializer_list` containing the elements to copy.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor specifies an empty controlled sequence. The third constructor specifies a copy of the sequence by moving *Right* The fourth through eighth constructors use an `initializer_list` to specify the elements to copy. The ninth constructor inserts the sequence of element values `[first, last)`.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *bucket_count*, if present; otherwise it's a default value described here as the implementation-defined value `N0`.

The hash function object is the argument *Hash*, if present; otherwise it's `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it's `Comp()`.

The allocator object is the argument *Al*, if present; otherwise, it's `Alloc()`.

value_type

The type of an element.

C++

```
typedef Key value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_set_value_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_set<char> Myset;
int main()
{
    Myset c1;
```

```
    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin(); it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[d] [c] [b] [a]
```

unordered_multiset Class

Article • 08/17/2021

The class template describes an object that controls a varying-length sequence of elements of type `const Key`. The sequence is weakly ordered by a hash function, which partitions the sequence into an ordered set of subsequences called buckets. Within each bucket a comparison function determines whether any pair of elements has equivalent ordering. Each element serves as both a sort key and a value. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations that can be independent of the number of elements in the sequence (constant time), at least when all buckets are of roughly equal length. In the worst case, when all of the elements are in one bucket, the number of operations is proportional to the number of elements in the sequence (linear time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

Syntax

C++

```
template <class Key,
    class Hash = std::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key>>
class unordered_multiset;
```

Parameters

Key
The key type.

Hash
The hash function object type.

Pred
The equality comparison function object type.

Alloc
The allocator class.

Members

Type Definition	Description
allocator_type	The type of an allocator for managing storage.
const_iterator	The type of a constant iterator for the controlled sequence.
const_local_iterator	The type of a constant bucket iterator for the controlled sequence.
const_pointer	The type of a constant pointer to an element.
const_reference	The type of a constant reference to an element.
difference_type	The type of a signed distance between two elements.
hasher	The type of the hash function.
iterator	The type of an iterator for the controlled sequence.
key_equal	The type of the comparison function.
key_type	The type of an ordering key.
local_iterator	The type of a bucket iterator for the controlled sequence.
pointer	The type of a pointer to an element.
reference	The type of a reference to an element.
size_type	The type of an unsigned distance between two elements.
value_type	The type of an element.

Member Function	Description
begin	Designates the beginning of the controlled sequence.
bucket	Gets the bucket number for a key value.
bucket_count	Gets the number of buckets.
bucket_size	Gets the size of a bucket.
cbegin	Designates the beginning of the controlled sequence.
cend	Designates the end of the controlled sequence.
clear	Removes all elements.
contains C++20	Checks if there's an element with the specified key.

Member Function	Description
count	Finds the number of elements matching a specified key.
emplace	Adds an element constructed in place.
emplace_hint	Adds an element constructed in place, with hint.
empty	Tests whether no elements are present.
end	Designates the end of the controlled sequence.
equal_range	Finds range that matches a specified key.
erase	Removes elements at specified positions.
find	Finds an element that matches a specified key.
get_allocator	Gets the stored allocator object.
hash_function	Gets the stored hash function object.
insert	Adds elements.
key_eq	Gets the stored comparison function object.
load_factor	Counts the average elements per bucket.
max_bucket_count	Gets the maximum number of buckets.
max_load_factor	Gets or sets the maximum elements per bucket.
max_size	Gets the maximum size of the controlled sequence.
rehash	Rebuilds the hash table.
size	Counts the number of elements.
swap	Swaps the contents of two containers.
unordered_multiset	Constructs a container object.

Operator	Description
unordered_multiset::operator=	Copies a hash table.

Remarks

The object orders the sequence it controls by calling two stored objects, a comparison function object of type unordered_multiset::key_equal and a hash function object of

type unordered_multiset::hasher. You access the first stored object by calling the member function unordered_multiset::key_eq `()`; and you access the second stored object by calling the member function unordered_multiset::hash_function `()`. Specifically, for all values `X` and `Y` of type `Key`, the call `key_eq()(X, Y)` returns true only if the two argument values have equivalent ordering; the call `hash_function()(keyval)` yields a distribution of values of type `size_t`. Unlike class template unordered_set Class, an object of type `unordered_multiset` does not ensure that `key_eq()(X, Y)` is always false for any two elements of the controlled sequence. (Keys need not be unique.)

The object also stores a maximum load factor, which specifies the maximum desired average number of elements per bucket. If inserting an element causes unordered_multiset::load_factor `()` to exceed the maximum load factor, the container increases the number of buckets and rebuilds the hash table as needed.

The actual order of elements in the controlled sequence depends on the hash function, the comparison function, the order of insertion, the maximum load factor, and the current number of buckets. You cannot in general predict the order of elements in the controlled sequence. You can always be assured, however, that any subset of elements that have equivalent ordering are adjacent in the controlled sequence.

The object allocates and frees storage for the sequence it controls through a stored allocator object of type unordered_multiset::allocator_type. Such an allocator object must have the same external interface as an object of type `allocator`. Note that the stored allocator object is not copied when the container object is assigned.

Requirements

Header: <unordered_set>

Namespace: std

unordered_multiset::allocator_type

The type of an allocator for managing storage.

```
C++

typedef Alloc allocator_type;
```

Remarks

The type is a synonym for the template parameter `Alloc`.

Example

C++

```
// std__unordered_set__unordered_multiset_allocator_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;

    return (0);
}
```

Output

```
al == std::allocator() is true
```

unordered_multiset::begin

Designates the beginning of the controlled sequence or a bucket.

C++

```
iterator begin();

const_iterator begin() const;

local_iterator begin(size_type nbucket);

const_local_iterator begin(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The first two member functions return a forward iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). The last two member functions return a forward iterator that points at the first element of bucket *nbucket* (or just beyond the end of an empty bucket).

Example

C++

```
// std__unordered_set__unordered_multiset_begin.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect first two items "[c] [b]"
    Myset::iterator it2 = c1.begin();
    std::cout << "[" << *it2 << "] ";
    ++it2;
    std::cout << "[" << *it2 << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[c] [b]
[a]
```

unordered_multiset::bucket

Gets the bucket number for a key value.

C++

```
size_type bucket(const Key& keyval) const;
```

Parameters

keyval
The key value to map.

Remarks

The member function returns the bucket number currently corresponding to the key value `keyval`.

Example

C++

```
// std__unordered_set__unordered_multiset_bucket.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
```

```
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multiset::bucket_count

Gets the number of buckets.

C++

```
size_type bucket_count() const;
```

Remarks

The member function returns the current number of buckets.

Example

C++

```
// std__unordered_set__unordered_multiset_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
```

```
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
```

```
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multiset::bucket_size

Gets the size of a bucket

C++

```
size_type bucket_size(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The member functions returns the size of bucket number *nbucket*.

Example

C++

```
// std__unordered_set__unordered_multiset_bucket_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;
```

```
    // display buckets for keys
    Myset::size_type bs = c1.bucket('a');
    std::cout << "bucket('a') == " << bs << std::endl;
    std::cout << "bucket_size(" << bs << ") == " << c1.bucket_size(bs)
        << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket('a') == 7
bucket_size(7) == 1
```

unordered_multiset::cbegin

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return Value

A `const` forward-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
```

```
auto i2 = Container.cbegin();
// i2 is Container<T>::const_iterator
```

unordered_multiset::cend

Returns a **const** iterator that addresses the location just beyond the last element in a range.

```
C++

const_iterator cend() const;
```

Return Value

A **const** forward-access iterator that points just beyond the end of the range.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- **const**) container of any kind that supports `end()` and `cend()`.

```
C++

auto i1 = Container.end();
// i1 is Container<T>::iterator

auto i2 = Container.cend();
// i2 is Container<T>::const_iterator
```

The value returned by `cend` should not be dereferenced.

unordered_multiset::clear

Removes all elements.

```
```

C++

```
void clear();
```

Remarks

The member function calls unordered_multiset::erase(unordered_multiset::begin(), unordered_multiset::end()).

Example

C++

```
// std__unordered_set__unordered_multiset_clear.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a] "
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d] "
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
```

```
    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

unordered_multiset::const_iterator

The type of a constant iterator for the controlled sequence.

C++

```
typedef T1 const_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for the controlled sequence. It is described here as a synonym for the implementation-defined type `T1`.

Example

C++

```
// std__unordered_set__unordered_multiset_const_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
```

```
    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

unordered_multiset::const_local_iterator

The type of a constant bucket iterator for the controlled sequence.

C++

```
typedef T5 const_local_iterator;
```

Remarks

The type describes an object that can serve as a constant forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T5`.

Example

C++

```
// std__unordered_set__unordered_multiset_const_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
```

```
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a]
```

unordered_multiset::const_pointer

The type of a constant pointer to an element.

C++

```
typedef Alloc::const_pointer const_pointer;
```

Remarks

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_const_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
```

```
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Myset::const_pointer p = &*it;
        std::cout << "[" << *p << "] ";
        }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

unordered_multiset::const_reference

The type of a constant reference to an element.

C++

```
typedef Alloc::const_reference const_reference;
```

Remarks

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_const_reference.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
```

```
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Myset::const_reference ref = *it;
        std::cout << "[" << ref << "] ";
        }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

unordered_multiset::contains

Checks if there's an element with the specified key in the `unordered_multiset`.

C++

```
bool contains(const Key& key) const;
template<class K> bool contains(const K& key) const;
```

Parameters

K
The type of the key.

key
The element's key value to look for.

Return Value

`true` if the element is found in the container; `false` otherwise.

Remarks

`contains()` is new in C++20. To use it, specify the /std:c++20 or later compiler option.

`template<class K> bool contains(const K& key) const` only participates in overload resolution if `key_compare` is transparent.

Example

C++

```
// Requires /std:c++20 or /std:c++latest
#include <unordered_set>
#include <iostream>

int main()
{
    std::unordered_multiset<int> theUnorderedMultiset = { 1, 2, 3 };

    std::cout << std::boolalpha; // so booleans show as 'true' or 'false'
    std::cout << theUnorderedMultiset.contains(1) << '\n';
    std::cout << theUnorderedMultiset.contains(4) << '\n';

    return 0;
}
```

Output

```
true
false
```

unordered_multiset::count

Finds the number of elements matching a specified key.

C++

```
size_type count(const Key& keyval) const;
```

Parameters

keyval
Key value to search for.

Remarks

The member function returns the number of elements in the range delimited by unordered_multiset::equal_range `(keyval)`.

Example

C++

```
// std__unordered_set__unordered_multiset_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "count('A') == " << c1.count('A') << std::endl;
    std::cout << "count('b') == " << c1.count('b') << std::endl;
    std::cout << "count('C') == " << c1.count('C') << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
count('A') == 0
count('b') == 1
count('C') == 0
```

unordered_multiset::difference_type

The type of a signed distance between two elements.

C++

```
typedef T3 difference_type;
```

Remarks

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the implementation-defined type `T3`.

Example

C++

```
// std__unordered_set__unordered_multiset_difference_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // compute positive difference
    Myset::difference_type diff = 0;
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        ++diff;
    std::cout << "end()-begin() == " << diff << std::endl;

    // compute negative difference
    diff = 0;
    for (Myset::const_iterator it = c1.end();
        it != c1.begin(); --it)
        --diff;
    std::cout << "begin()-end() == " << diff << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
end()-begin() == 3
begin()-end() == -3
```

unordered_multiset::emplace

Inserts an element constructed in place (no copy or move operations are performed).

C++

```
template <class... Args>
iterator emplace(Args&&... args);
```

Parameters

args
The arguments forwarded to construct an element to be inserted into the unordered_multiset.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see multiset::emplace.

unordered_multiset::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

C++

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

args
The arguments forwarded to construct an element to be inserted into the unordered_multiset.

where
A hint regarding the place to start searching for the correct point of insertion.

Return Value

An iterator to the newly inserted element.

Remarks

No references to container elements are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion, if an exception is thrown but does not occur in the container's hash function, the container is not modified. If the exception is thrown in the hash function, the result is undefined.

For a code example, see set::emplace_hint.

unordered_multiset::empty

Tests whether no elements are present.

C++

```
bool empty() const;
```

Remarks

The member function returns true for an empty controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_empty.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
```

```
size == 2
empty() == false
```

unordered_multiset::end

Designates the end of the controlled sequence.

C++

```
iterator end();
const_iterator end() const;
local_iterator end(size_type nbucket);
const_local_iterator end(size_type nbucket) const;
```

Parameters

nbucket
The bucket number.

Remarks

The first two member functions return a forward iterator that points just beyond the end of the sequence. The last two member functions return a forward iterator that points just beyond the end of bucket *nbucket*.

Example

C++

```
// std__unordered_set__unordered_multiset_end.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
```

```
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect last two items "[a] [b]"
    Myset::iterator it2 = c1.end();
    --it2;
    std::cout << "[" << *it2 << "] ";
    --it2;
    std::cout << "[" << *it2 << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::const_local_iterator lit = c1.end(c1.bucket('a'));
    --lit;
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a] [b]
[a]
```

unordered_multiset::equal_range

Finds range that matches a specified key.

C++

```
std::pair<iterator, iterator>
    equal_range(const Key& keyval);

std::pair<const_iterator, const_iterator>
    equal_range(const Key& keyval) const;
```

Parameters

keyval
Key value to search for.

Remarks

The member function returns a pair of iterators `X` such that `[X.first, X.second)` delimits just those elements of the controlled sequence that have equivalent ordering with *keyval*. If no such elements exist, both iterators are `end()`.

Example

C++

```
// std__unordered_set__unordered_multiset_equal_range.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // display results of failed search
    std::pair<Myset::iterator, Myset::iterator> pair1 =
        c1.equal_range('x');
    std::cout << "equal_range('x'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "] ";
    std::cout << std::endl;

    // display results of successful search
    pair1 = c1.equal_range('b');
    std::cout << "equal_range('b'):";
    for (; pair1.first != pair1.second; ++pair1.first)
        std::cout << "[" << *pair1.first << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
equal_range('x'):
```

```
equal_range('b'): [b]
```

unordered_multiset::erase

Removes an element or a range of elements in a unordered_multiset from specified positions or removes elements that match a specified key.

```
C++

iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

Where
Position of the element to be removed.

First
Position of the first element to be removed.

Last
Position just beyond the last element to be removed.

Key
The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the unordered_multiset if no such element exists.

For the third member function, returns the number of elements that have been removed from the unordered_multiset.

Remarks

For a code example, see set::erase.

unordered_multiset::find

Finds an element that matches a specified key.

C++

```
const_iterator find(const Key& keyval) const;
```

Parameters

keyval
Key value to search for.

Remarks

The member function returns unordered_multiset::equal_range`(keyval).first`.

Example

C++

```
// std__unordered_set__unordered_multiset_find.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // try to find and fail
    std::cout << "find('A') == "
        << std::boolalpha << (c1.find('A') != c1.end()) << std::endl;
```

```
    // try to find and succeed
    Myset::iterator it = c1.find('b');
    std::cout << "find('b') == "
        << std::boolalpha << (it != c1.end())
        << ": [" << *it << "] " << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
find('A') == false
find('b') == true: [b]
```

unordered_multiset::get_allocator

Gets the stored allocator object.

C++

```
Alloc get_allocator() const;
```

Remarks

The member function returns the stored allocator object.

Example

C++

```
// std__unordered_set__unordered_multiset_get_allocator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
typedef std::allocator<std::pair<const char, int> > Myalloc;
int main()
{
    Myset c1;

    Myset::allocator_type al = c1.get_allocator();
    std::cout << "al == std::allocator() is "
        << std::boolalpha << (al == Myalloc()) << std::endl;
```

```
    return (0);
}
```

Output

```
al == std::allocator() is true
```

unordered_multiset::hash_function

Gets the stored hash function object.

C++

```
Hash hash_function() const;
```

Remarks

The member function returns the stored hash function object.

Example

C++

```
// std__unordered_set__unordered_multiset_hash_function.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multiset::hasher

The type of the hash function.

C++

```
typedef Hash hasher;
```

Remarks

The type is a synonym for the template parameter `Hash`.

Example

C++

```
// std__unordered_set__unordered_multiset_hasher.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::hasher hfn = c1.hash_function();
    std::cout << "hfn('a') == " << hfn('a') << std::endl;
    std::cout << "hfn('b') == " << hfn('b') << std::endl;

    return (0);
}
```

Output

```
hfn('a') == 1630279
hfn('b') == 1647086
```

unordered_multiset::insert

Inserts an element or a range of elements into an unordered_multiset.

C++

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);

// (2) single element, perfect forwarded
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);

// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);

// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);

// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);

// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Val

The value of an element to be inserted into the unordered_multiset.

Where

The place to start searching for the correct point of insertion.

ValTy

Template parameter that specifies the argument type that the unordered_multiset can use to construct an element of value_type, and perfect-forwards *Val* as an argument.

First

The position of the first element to be copied.

Last
The position just beyond the last element to be copied.

InputIterator
Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.

IList
The initializer_list from which to copy the elements.

Return Value

The single-element-insert member functions, (1) and (2), return an iterator to the position where the new element was inserted into the unordered_multiset.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the unordered_multiset.

Remarks

No pointers or references are invalidated by this function, but it may invalidate all iterators to the container.

During the insertion of just one element, if an exception is thrown but does not occur in the container's hash function, the container's state is not modified. If the exception is thrown in the hash function, the result is undefined. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

The value_type of a container is a typedef that belongs to the container, and, for set, `unordered_multiset<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into an unordered_multiset that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, *Last* does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `m.insert(v.begin(), v.end());` inserts all elements of `v` into `m`.

The initializer list member function (6) uses an initializer_list to copy elements into the unordered_multiset.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see unordered_multiset::emplace and unordered_multiset::emplace_hint.

For a code example, see multiset::insert.

unordered_multiset::iterator

A type that provides a constant forward iterator that can read elements in an unordered_multiset.

```
C++
```

```
typedef implementation-defined iterator;
```

Example

See the example for begin for an example of how to declare and use an **iterator**.

unordered_multiset::key_eq

Gets the stored comparison function object.

```
C++
```

```
Pred key_eq() const;
```

Remarks

The member function returns the stored comparison function object.

Example

```
C++
```

```
// std__unordered_set__unordered_multiset_key_eq.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
```

```
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multiset::key_equal

The type of the comparison function.

C++

```
typedef Pred key_equal;
```

Remarks

The type is a synonym for the template parameter `Pred`.

Example

C++

```
// std__unordered_set__unordered_multiset_key_equal.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    Myset::key_equal cmpfn = c1.key_eq();
    std::cout << "cmpfn('a', 'a') == "
        << std::boolalpha << cmpfn('a', 'a') << std::endl;
    std::cout << "cmpfn('a', 'b') == "
        << std::boolalpha << cmpfn('a', 'b') << std::endl;

    return (0);
}
```

Output

```
cmpfn('a', 'a') == true
cmpfn('a', 'b') == false
```

unordered_multiset::key_type

The type of an ordering key.

C++

```
typedef Key key_type;
```

Remarks

The type is a synonym for the template parameter `Key`.

Example

C++

```
// std__unordered_set__unordered_multiset_key_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);
```

```
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[d] [c] [b] [a]
```

unordered_multiset::load_factor

Counts the average elements per bucket.

C++

```
float load_factor() const;
```

Remarks

The member function returns `(float)` unordered_multiset::size `() /` `(float)` unordered_multiset::bucket_count `()`, the average number of elements per bucket.

Example

C++

```
// std__unordered_set__unordered_multiset_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');
```

```
    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

unordered_multiset::local_iterator

The type of a bucket iterator.

```
C++

typedef T4 local_iterator;
```

Remarks

The type describes an object that can serve as a forward iterator for a bucket. It is described here as a synonym for the implementation-defined type `T4`.

Example

C++

```
// std__unordered_set__unordered_multiset_local_iterator.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect bucket containing 'a'
    Myset::local_iterator lit = c1.begin(c1.bucket('a'));
    std::cout << "[" << *lit << "] ";

    return (0);
}
```

Output

```
[c] [b] [a]
[a]
```

unordered_multiset::max_bucket_count

Gets the maximum number of buckets.

C++

```
size_type max_bucket_count() const;
```

Remarks

The member function returns the maximum number of buckets currently permitted.

Example

C++

```
// std__unordered_set__unordered_multiset_max_bucket_count.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
```

```
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multiset::max_load_factor

Gets or sets the maximum elements per bucket.

C++

```
float max_load_factor() const;

void max_load_factor(float factor);
```

Parameters

factor
The new maximum load factor.

Remarks

The first member function returns the stored maximum load factor. The second member function replaces the stored maximum load factor with *factor*.

Example

C++

```
// std__unordered_set__unordered_multiset_max_load_factor.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_bucket_count() == "
        << c1.max_bucket_count() << std::endl;
    std::cout << "max_load_factor() == "
        << c1.max_load_factor() << std::endl;
    std::cout << std::endl;
```

```
    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_bucket_count() == 8
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_bucket_count() == 128
max_load_factor() == 0.1
```

unordered_multiset::max_size

Gets the maximum size of the controlled sequence.

C++

```
size_type max_size() const;
```

Remarks

The member function returns the length of the longest sequence that the object can control.

Example

C++

```
// std__unordered_set__unordered_multiset_max_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
```

```
    Myset c1;

    std::cout << "max_size() == " << c1.max_size() << std::endl;

    return (0);
}
```

Output

```
max_size() == 4294967295
```

unordered_multiset::operator=

Copies a hash table.

C++

```
unordered_multiset& operator=(const unordered_multiset& right);

unordered_multiset& operator=(unordered_multiset&& right);
```

Parameters

right
The unordered_multiset being copied into the `unordered_multiset`.

Remarks

After erasing any existing elements in an `unordered_multiset`, `operator=` either copies or moves the contents of *right* into the `unordered_multiset`.

Example

C++

```
// unordered_multiset_operator_as.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

int main( )
{
    using namespace std;
```

```
    unordered_multiset<int> v1, v2, v3;
    unordered_multiset<int>::iterator iter;

    v1.insert(10);

    cout << "v1 = " ;
    for (iter = v1.begin(); iter != v1.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    v2 = v1;
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    // move v1 into v2
    v2.clear();
    v2 = move(v1);
    cout << "v2 = ";
    for (iter = v2.begin(); iter != v2.end(); iter++)
        cout << *iter << " ";
    cout << endl;
}
```

unordered_multiset::pointer

The type of a pointer to an element.

C++

```
typedef Alloc::pointer pointer;
```

Remarks

The type describes an object that can serve as a pointer to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_pointer.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>
```

```
typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Myset::key_type key = *it;
        Myset::pointer p = &key;
        std::cout << "[" << *p << "] ";
        }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

unordered_multiset::reference

The type of a reference to an element.

C++

```
typedef Alloc::reference reference;
```

Remarks

The type describes an object that can serve as a reference to an element of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_reference.cpp
// compile with: /EHsc
#include <unordered_set>
```

```cpp
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::iterator it = c1.begin();
        it != c1.end(); ++it)
        {
        Myset::key_type key = *it;
        Myset::reference ref = key;
        std::cout << "[" << ref << "] ";
        }
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
```

unordered_multiset::rehash

Rebuilds the hash table.

C++

```cpp
void rehash(size_type nbuckets);
```

Parameters

nbuckets
The requested number of buckets.

Remarks

The member function alters the number of buckets to be at least *nbuckets* and rebuilds the hash table as needed.

Example

C++

```
// std__unordered_set__unordered_multiset_rehash.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // inspect current parameters
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

    // change max_load_factor and redisplay
    c1.max_load_factor(0.10f);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;
    std::cout << std::endl;

    // rehash and redisplay
    c1.rehash(100);
    std::cout << "bucket_count() == " << c1.bucket_count() << std::endl;
    std::cout << "load_factor() == " << c1.load_factor() << std::endl;
    std::cout << "max_load_factor() == " << c1.max_load_factor() <<
std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 4

bucket_count() == 8
load_factor() == 0.375
max_load_factor() == 0.1

bucket_count() == 128
load_factor() == 0.0234375
max_load_factor() == 0.1
```

unordered_multiset::size

Counts the number of elements.

C++

```
size_type size() const;
```

Remarks

The member function returns the length of the controlled sequence.

Example

C++

```
// std__unordered_set__unordered_multiset_size.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
```

```
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // clear the container and reinspect
    c1.clear();
    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;
    std::cout << std::endl;

    c1.insert('d');
    c1.insert('e');

    // display contents "[e] [d]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    std::cout << "size == " << c1.size() << std::endl;
    std::cout << "empty() == " << std::boolalpha << c1.empty() << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
size == 0
empty() == true

[e] [d]
size == 2
empty() == false
```

unordered_multiset::size_type

The type of an unsigned distance between two elements.

C++

```
typedef T2 size_type;
```

Remarks

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the implementation-defined type `T2`.

Example

C++

```
// std__unordered_set__unordered_multiset_size_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;
    Myset::size_type sz = c1.size();

    std::cout << "size == " << sz << std::endl;

    return (0);
}
```

Output

```
size == 0
```

unordered_multiset::swap

Swaps the contents of two containers.

C++

```
void swap(unordered_multiset& right);
```

Parameters

right
The container to swap with.

Remarks

The member function swaps the controlled sequences between `*this` and *right*. If unordered_multiset::get_allocator`() == right.get_allocator()`, it does so in constant time, it throws an exception only as a result of copying the stored traits object of type `Tr`, and it invalidates no references, pointers, or iterators that designate elements in the

two controlled sequences. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Example

C++

```
// std__unordered_set__unordered_multiset_swap.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    Myset c2;

    c2.insert('d');
    c2.insert('e');
    c2.insert('f');

    c1.swap(c2);

    // display contents "[f] [e] [d]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    swap(c1, c2);

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[f] [e] [d]
[c] [b] [a]
```

unordered_multiset::unordered_multiset

Constructs a container object.

C++

```
unordered_multiset(
    const unordered_multiset& Right);

explicit unordered_multiset(
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());

unordered_multiset(
    unordered_multiset&& Right);

unordered_set(
    initializer_list<Type> IList);

unordered_set(
    initializer_list<Typ> IList,
    size_type Bucket_count);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key);

unordered_set(
    initializer_list<Type> IList,
    size_type Bucket_count,
    const Hash& Hash,
    const Key& Key,
    const Allocator& Al);

template <class InputIterator>
```

```
unordered_multiset(
    InputIterator First,
    InputIterator Last,
    size_type Bucket_count = N0,
    const Hash& Hash = Hash(),
    const Comp& Comp = Comp(),
    const Allocator& Al = Alloc());
```

Parameters

InputIterator
The iterator type.

Al
The allocator object to store.

Comp
The comparison function object to store.

Hash
The hash function object to store.

Bucket_count
The minimum number of buckets.

Right
The container to copy.

IList
The initializer_list from which to copy.

Remarks

The first constructor specifies a copy of the sequence controlled by *Right*. The second constructor specifies an empty controlled sequence. The third constructor inserts the sequence of element values `[First, Last)`. The fourth constructor specifies a copy of the sequence by moving *Right*.

All constructors also initialize several stored values. For the copy constructor, the values are obtained from *Right*. Otherwise:

The minimum number of buckets is the argument *Bucket_count*, if present; otherwise it is a default value described here as the implementation-defined value `N0`.

The hash function object is the argument *Hash*, if present; otherwise it is `Hash()`.

The comparison function object is the argument *Comp*, if present; otherwise it is `Comp()`.

The allocator object is the argument *Al*, if present; otherwise, it is `Alloc()`.

unordered_multiset::value_type

The type of an element.

```
C++
```

```
typedef Key value_type;
```

Remarks

The type describes an element of the controlled sequence.

Example

```
C++
```

```
// std__unordered_set__unordered_multiset_value_type.cpp
// compile with: /EHsc
#include <unordered_set>
#include <iostream>

typedef std::unordered_multiset<char> Myset;
int main()
{
    Myset c1;

    c1.insert('a');
    c1.insert('b');
    c1.insert('c');

    // display contents "[c] [b] [a]"
    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
    std::cout << std::endl;

    // add a value and reinspect
    Myset::key_type key = 'd';
    Myset::value_type val = key;
    c1.insert(val);

    for (Myset::const_iterator it = c1.begin();
        it != c1.end(); ++it)
        std::cout << "[" << *it << "] ";
```

```
    std::cout << std::endl;

    return (0);
}
```

Output

```
[c] [b] [a]
[d] [c] [b] [a]
```

See also

<unordered_set>

Containers

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

`<utility>`

Article • 11/15/2024

Defines C++ Standard Library types, functions, and operators that help to construct and manage pairs of objects, which are useful whenever two objects should be treated as if they were one.

Requirements

Header: `<utility>`

Namespace: `std`

Remarks

Pairs are widely used in the C++ Standard Library. They're required both as the arguments and return values for various functions and as element types for associative containers like map and multimap.

Members

Classes

Expand table

Type	Description
chars_format	Floating-point format for primitive numerical conversion.
tuple_element	Wraps the type of a `pair` element.
tuple_size	Wraps a `pair` element count.

Objects

Expand table

Template	Description
index_sequence	An alias template defined for the common case where `T` is `std::size_t`

Template	Description
index_sequence_for	Helper alias template to convert any type parameter pack into an index sequence of the same length
make_index_sequence	Helper alias template to simplify the creation of a `std::index_sequence` type.
make_integer_sequence	Helper alias template to simplify the creation of a `std::integer_sequence` type.

Functions

Expand table

Function	Description
as_const	Returns type.
declval	Shorthand expression evaluation.
exchange	Assigns a new value to an object and returns its old value.
forward	Preserves the reference type (either `lvalue` or `rvalue`) of the argument from being obscured by perfect forwarding.
from_chars	
get	A function that gets an element from a `pair` object.
make_pair	A template helper function used to construct objects of type `pair`, where the component types are based on the data types passed as parameters.
move	Returns the passed in argument as an `rvalue` reference.
move_if_noexcept	
swap	Exchanges the elements of two `pair` objects.
to_chars	Converts value into a character string.

Operators

Expand table

Operator	Description
operator!=	Tests if the pair object on the left side of the operator isn't equal to the pair object on the right side.
operator==	Tests if the pair object on the left side of the operator is equal to the pair object on the right side.
operator<	Tests if the pair object on the left side of the operator is less than the pair object on the right side.
operator<=	Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.
operator>	Tests if the pair object on the left side of the operator is greater than the pair object on the right side.
operator>=	Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

Structs

Expand table

Struct	Description
from_chars_result	A struct used for `from_chars`.
identity	A struct that provides a type definition as the template parameter.
in_place_t, in_place_type_t, in_place_index_t	Indicates how to create an object in place.
integer_sequence	Represents an integer sequence.
pair	A type that provides for the ability to treat two objects as a single object.
piecewise_construct_t	A type used to keep separate constructor and function overloading.
to_chars_result	A struct used for `to_chars`.

See also

Header Files Reference
Thread Safety in the C++ Standard Library

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

`<utility>` functions

Article • 10/17/2022

`as_const`

```
C++
```

```
template <class T> constexpr add_const_t<T>& as_const(T& t) noexcept;
template <class T> void as_const(const T&&) = delete;
```

Return Value

Returns *`T`*.

`declval`

```
C++
```

```
template <class T> add_rvalue_reference_t<T> declval() noexcept;  // as
unevaluated operand
```

`exchange`

`(C++14)` Assigns a new value to an object and returns its old value.

```
C++
```

```
template <class T, class Other = T>
    T exchange(T& val, Other&& new_val)
```

Parameters

`val`

The object that will receive the value of `new_val`.

`new_val`

The object whose value is copied or moved into `val`.

Remarks

For complex types, `exchange` avoids copying the old value when a move constructor is available, avoids copying the new value if it's a temporary object or is moved, and accepts any type as the new value, using any available converting assignment operator. The exchange function is different from std::swap in that the left argument isn't moved or copied to the right argument.

Example

The following example shows how to use `exchange`. In the real world, `exchange` is most useful with large objects that are expensive to copy:

C++

```
#include <utility>
#include <iostream>

using namespace std;

struct C
{
   int i;
   //...
};

int main()
{
   // Use brace initialization
   C c1{ 1 };
   C c2{ 2 };
   C result = exchange(c1, c2);
   cout << "The old value of c1 is: " << result.i << endl;
   cout << "The new value of c1 after exchange is: " << c1.i << endl;

   return 0;
}
```

Output

```
The old value of c1 is: 1
The new value of c1 after exchange is: 2
```

`forward`

Conditionally casts its argument to an rvalue reference if the argument is an rvalue or rvalue reference. This restores the rvalue-ness of an argument to the forwarding function in support of perfect forwarding.

```
C++
```

```
template <class Type>    // accepts lvalues
    constexpr Type&& forward(typename remove_reference<Type>::type& Arg)
noexcept

template <class Type>    // accepts everything else
    constexpr Type&& forward(typename remove_reference<Type>::type&& Arg)
noexcept
```

Parameters

Type

The type of the value passed in *Arg*, which might be different than the type of *Arg*. Typically determined by a template argument of the forwarding function.

Arg

The argument to cast.

Return Value

Returns an rvalue reference to *Arg* if the value passed in *Arg* was originally an rvalue or a reference to an rvalue; otherwise, returns *Arg* without modifying its type.

Remarks

You must specify an explicit template argument to call `forward`.

`forward` doesn't forward its argument. Instead, by conditionally casting its argument to an rvalue reference if it was originally an rvalue or rvalue reference, `forward` enables the compiler to perform overload resolution with knowledge of the forwarded argument's original type. The apparent type of an argument to a forwarding function might be different than its original type—for example, when an rvalue is used as an argument to a function and is bound to a parameter name; having a name makes it an lvalue, with whatever value actually exists as an rvalue— `forward` restores the rvalue-ness of the argument.

Restoring the rvalue-ness of an argument's original value to do overload resolution is known as *perfect forwarding*. Perfect forwarding enables a template function to accept an argument of either reference type and to restore its rvalue-ness when it's necessary for correct overload resolution. By using perfect forwarding, you can preserve move semantics for rvalues and avoid having to provide overloads for functions that vary only by the reference type of their arguments.

from_chars

C++

```
from_chars_result from_chars(const char* first, const char* last, see below&
value, int base = 10);

from_chars_result from_chars(const char* first, const char* last, float&
value, chars_format fmt = chars_format::general);

from_chars_result from_chars(const char* first, const char* last, double&
value, chars_format fmt = chars_format::general);

from_chars_result from_chars(const char* first, const char* last, long
double& value, chars_format fmt = chars_format::general);
```

`get`

Gets an element from a `pair` object by index position, or by type.

C++

```
// get reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
    constexpr tuple_element_t<Index, pair<T1, T2>>&
    get(pair<T1, T2>& Pr) noexcept;

// get reference to element T1 in pair Pr
template <class T1, class T2>
    constexpr T1& get(pair<T1, T2>& Pr) noexcept;

// get reference to element T2 in pair Pr
template <class T2, class T1>
    constexpr T2& get(pair<T1, T2>& Pr) noexcept;

// get const reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
    constexpr const tuple_element_t<Index, pair<T1, T2>>&
    get(const pair<T1, T2>& Pr) noexcept;
```

```
// get const reference to element T1 in pair Pr
template <class T1, class T2>
    constexpr const T1& get(const pair<T1, T2>& Pr) noexcept;

// get const reference to element T2 in pair Pr
template <class T2, class T1>
    constexpr const T2& get(const pair<T1, T2>& Pr) noexcept;

// get rvalue reference to element at Index in pair Pr
template <size_t Index, class T1, class T2>
    constexpr tuple_element_t<Index, pair<T1, T2>>&&
    get(pair<T1, T2>&& Pr) noexcept;

// get rvalue reference to element T1 in pair Pr
template <class T1, class T2>
    constexpr T1&& get(pair<T1, T2>&& Pr) noexcept;

// get rvalue reference to element T2 in pair Pr
template <class T2, class T1>
    constexpr T2&& get(pair<T1, T2>&& Pr) noexcept;
```

Parameters

Index

The 0-based index of the chosen element.

T1

The type of the first pair element.

T2

The type of the second pair element.

pr

The pair to select from.

Remarks

The template functions each return a reference to an element of its `pair` argument.

For the indexed overloads, if the value of *Index* is 0 the functions return `pr.first` and if the value of *Index* is 1 the functions return `pr.second`. The type `RI` is the type of the returned element.

For the overloads that don't have an `Index` parameter, the element to return is deduced by the type argument. Calling `get<T>(Tuple)` will produce a compiler error if *pr* contains more or less than one element of type `T`.

Example

C++

```
#include <utility>
#include <iostream>
using namespace std;
int main()
{

    typedef pair<int, double> MyPair;

    MyPair c0(9, 3.14);

    // get elements by index
    cout << " " << get<0>(c0);
    cout << " " << get<1>(c0) << endl;

    // get elements by type (C++14)
    MyPair c1(1, 0.27);
    cout << " " << get<int>(c1);
    cout << " " << get<double>(c1) << endl;
}
```

Output

```
9 3.14
1 0.27
```

index_sequence

C++

```
template<size_t... I>
    using index_sequence = integer_sequence<size_t, I...>;
```

index_sequence_for

C++

```
template<class... T>
    using index_sequence_for = make_index_sequence<sizeof...(T)>;
```

`make_index_sequence`

```
C++
```

```
template<size_t N>
    using make_index_sequence = make_integer_sequence<size_t, N>;
```

`make_integer_sequence`

```
C++
```

```
template<class T, T N>
    using make_integer_sequence = integer_sequence<T, see below >;
```

`make_pair`

A template function that you can use to construct objects of type `pair`, where the component types are automatically chosen based on the data types that are passed as parameters.

```
C++
```

```
template <class T, class U>
    pair<T, U> make_pair(T& Val1, U& Val2);

template <class T, class U>
    pair<T, U> make_pair(T& Val1, U&& Val2);

template <class T, class U>
    pair<T, U> make_pair(T&& Val1, U& Val2);

template <class T, class U>
    pair<T, U> make_pair(T&& Val1, U&& Val2);
```

Parameters

`Val1`

Value that initializes the first element of `pair`.

`Val2`

Value that initializes the second element of `pair`.

Return Value

The pair object that's constructed: `pair<T,U>(Val1, Val2)`.

Remarks

`make_pair` converts object of type reference_wrapper Class to reference types and converts decaying arrays and functions to pointers.

In the returned `pair` object, `T` is determined as follows:

- If the input type `T` is `reference_wrapper<X>`, the returned type `T` is `X&`.
- Otherwise, the returned type `T` is `decay<T>::type`. If decay Class isn't supported, the returned type `T` is the same as the input type `T`.

The returned type `U` is similarly determined from the input type `U`.

One advantage of `make_pair` is that the types of objects that are being stored are determined automatically by the compiler and don't have to be explicitly specified. Don't use explicit template arguments such as `make_pair<int, int>(1, 2)` when you use `make_pair` because it's verbose and adds complex rvalue reference problems that might cause compilation failure. For this example, the correct syntax would be `make_pair(1, 2)`

The `make_pair` helper function also makes it possible to pass two values to a function that requires a pair as an input parameter.

Example

For an example about how to use the helper function `make_pair` to declare and initialize a pair, see pair Structure.

`move`

Unconditionally casts its argument to an rvalue reference, and thereby signals that it can be moved if its type is move-enabled.

C++

```
template <class Type>
    constexpr typename remove_reference<Type>::type&& move(Type&& Arg)
noexcept;
```

Parameters

Type
A type deduced from the type of the argument passed in *Arg*, together with the reference collapsing rules.

Arg
The argument to cast. Although the type of *Arg* appears to be specified as an rvalue reference, `move` also accepts lvalue arguments because lvalue references can bind to rvalue references.

Return Value

`Arg` as an rvalue reference, whether or not its type is a reference type.

Remarks

The template argument *Type* isn't intended to be specified explicitly, but to be deduced from the type of the value passed in *Arg*. The type of *Type* is further adjusted according to the reference collapsing rules.

`move` doesn't move its argument. Instead, by unconditionally casting its argument—which might be an lvalue—to an rvalue reference, it enables the compiler to subsequently move, rather than copy, the value passed in *Arg* if its type is move-enabled. If its type isn't move-enabled, it's copied instead.

If the value passed in *Arg* is an lvalue—that is, it has a name or its address can be taken—it's invalidated when the move occurs. Don't refer to the value passed in *Arg* by its name or address after it's been moved.

`move_if_noexcept`

C++

```
template <class T> constexpr conditional_t<
!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&,
T&&> move_if_noexcept(T& x) noexcept;
```

`swap`

Exchanges the elements of two type or pair Structure objects.

```C++
template <class T>
    void swap(T& left, T& right) noexcept(see below );
template <class T, size_t N>
    void swap(T (&left)[N], T (&right)[N])
noexcept(is_nothrow_swappable_v<T>);
template <class T, class U>
    void swap(pair<T, U>& left, pair<T, U>& right);
```

Parameters

left

An object of type or type `pair`.

right

An object of type or type `pair`.

Remarks

One advantage of `swap` is that the types of objects that are being stored are determined automatically by the compiler and don't have to be explicitly specified. Don't use explicit template arguments such as `swap<int, int>(1, 2)` when you use `swap` because it's verbose and adds complex rvalue reference problems that might cause compilation failure.

`to_chars`

```C++
to_chars_result to_chars(char* first, char* last, see below value, int base
= 10);
to_chars_result to_chars(char* first, char* last, float value);
to_chars_result to_chars(char* first, char* last, double value);
to_chars_result to_chars(char* first, char* last, long double value);
to_chars_result to_chars(char* first, char* last, float value, chars_format
fmt);
to_chars_result to_chars(char* first, char* last, double value, chars_format
fmt);
to_chars_result to_chars(char* first, char* last, long double value,
chars_format fmt);
to_chars_result to_chars(char* first, char* last, float value, chars_format
fmt, int precision);
```

```
to_chars_result to_chars(char* first, char* last, double value, chars_format
fmt, int precision);
to_chars_result to_chars(char* first, char* last, long double value,
chars_format fmt, int precision);
```

Remarks

Converts value into a character string by filling the range `[first, last)`, where `[first, last)` is required to be a valid range.

`<utility>` operators

Article • 12/07/2021

> **Note**
>
> Operators using `Type&` are included under `namespace rel_ops`.

operator!=

Tests if the pair object on the left side of the operator is not equal to the pair object on the right side.

```
C++
```

```
template <class Type>
    constexpr bool operator!=(const Type& left, const Type& right);

template <class T, class U>
    constexpr bool operator!=(const pair<T, U>& left, const pair<T, U>&
right);
```

Parameters

left
An object of type `pair`.

right
An object of type `pair`.

Return Value

`true` if the pairs are not equal; `false` if the pairs are equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

C++

```
// utility_op_ne.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 1.11e-1 );
   p2 = make_pair ( 1000, 1.11e-3 );
   p3 = make_pair ( 10, 1.11e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

   if ( p1 != p2 )
      cout << "The pairs p1 and p2 are not equal." << endl;
   else
      cout << "The pairs p1 and p2 are equal." << endl;

   if ( p1 != p3 )
      cout << "The pairs p1 and p3 are not equal." << endl;
   else
      cout << "The pairs p1 and p3 are equal." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.111 ).
The pair p2 is: ( 1000, 0.00111 ).
The pair p3 is: ( 10, 0.111 ).

The pairs p1 and p2 are not equal.
The pairs p1 and p3 are equal.
```

operator==

Tests if the pair object on the left side of the operator is equal to the pair object on the right side.

C++

```
template <class T, class U>
constexpr bool operator==(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left
An object of type `pair`.

right
An object of type `pair`.

Return Value

`true` if the pairs are equal; `false` if the `pair`s are not equal.

Remarks

One pair is equal to another pair if each of their respective elements is equal. The function returns `left`. **first** == `right`. **first** && `left`. **second** == `right`. **second**. Two pairs are unequal if either the first or the second element of one is not equal to the corresponding element of the other pair.

Example

C++

```
// utility_op_eq.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 1.11e-1 );
   p2 = make_pair ( 1000, 1.11e-3 );
```

```
    p3 = make_pair ( 10, 1.11e-1 );

    cout.precision ( 3 );
    cout << "The pair p1 is: ( " << p1.first << ", "
         << p1.second << " )." << endl;
    cout << "The pair p2 is: ( " << p2.first << ", "
         << p2.second << " )." << endl;
    cout << "The pair p3 is: ( " << p3.first << ", "
         << p3.second << " )." << endl << endl;

    if ( p1 == p2 )
       cout << "The pairs p1 and p2 are equal." << endl;
    else
       cout << "The pairs p1 and p2 are not equal." << endl;

    if ( p1 == p3 )
       cout << "The pairs p1 and p3 are equal." << endl;
    else
       cout << "The pairs p1 and p3 are not equal." << endl;
}
```

`operator<`

Tests if the pair object on the left side of the operator is less than the pair object on the right side.

C++

```
template <class T, class U>
constexpr bool operator<(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left
An object of type `pair` on the left side of the operator.

right
An object of type `pair` on the right side of the operator.

Return Value

`true` if the `pair` on the left side of the operator is strictly less than the `pair` on the right side of the operator; otherwise **`false`**.

Remarks

The `left` `pair` object is said to be strictly less than the `right` `pair` object if *left* is less than and not equal to *right*.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

C++

```
// utility_op_lt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl << endl;

   if ( p1 < p2 )
      cout << "The pair p1 is less than the pair p2." << endl;
   else
      cout << "The pair p1 is not less than the pair p2." << endl;

   if ( p1 < p3 )
      cout << "The pair p1 is less than the pair p3." << endl;
   else
      cout << "The pair p1 is not less than the pair p3." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).

The pair p1 is less than the pair p2.
The pair p1 is not less than the pair p3.
```

`operator<=`

Tests if the pair object on the left side of the operator is less than or equal to the pair object on the right side.

C++

```
template <class Type>
constexpr bool operator<=(const Type& left, const Type& right);

template <class T, class U>
constexpr bool operator<=(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left
An object of type `pair` on the left side of the operator.

right
An object of type `pair` on the right side of the operator.

Return Value

`true` if the `pair` on the left side of the operator is less than or equal to the `pair` on the right side of the operator; otherwise **`false`**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

C++

```
// utility_op_le.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 <= p2 )
      cout << "The pair p1 is less than or equal to the pair p2." << endl;
   else
      cout << "The pair p1 is greater than the pair p2." << endl;

   if ( p1 <= p3 )
      cout << "The pair p1 is less than or equal to the pair p3." << endl;
   else
      cout << "The pair p1 is greater than the pair p3." << endl;

   if ( p1 <= p4 )
      cout << "The pair p1 is less than or equal to the pair p4." << endl;
   else
      cout << "The pair p1 is greater than the pair p4." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).
```

```
The pair p1 is less than or equal to the pair p2.
The pair p1 is greater than the pair p3.
The pair p1 is less than or equal to the pair p4.
```

`operator>`

Tests if the pair object on the left side of the operator is greater than the pair object on the right side.

C++

```
template <class Type>
constexpr bool operator>(const Type& left, const Type& right);

template <class T, class U>
constexpr bool operator>(const pair<T, U>& left, const pair<T, U>& right);
```

Parameters

left
An object of type `pair` on the left side of the operator.

right
An object of type `pair` on the right side of the operator.

Return Value

`true` if the `pair` on the left side of the operator is strictly greater than the `pair` on the right side of the operator; otherwise **`false`**.

Remarks

The `left` `pair` object is said to be strictly greater than the `right` `pair` object if *left* is greater than and not equal to *right*.

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

C++

```
// utility_op_gt.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 > p2 )
      cout << "The pair p1 is greater than the pair p2." << endl;
   else
      cout << "The pair p1 is not greater than the pair p2." << endl;

   if ( p1 > p3 )
      cout << "The pair p1 is greater than the pair p3." << endl;
   else
      cout << "The pair p1 is not greater than the pair p3." << endl;

   if ( p1 > p4 )
      cout << "The pair p1 is greater than the pair p4." << endl;
   else
      cout << "The pair p1 is not greater than the pair p4." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).
```

```
The pair p1 is not greater than the pair p2.
The pair p1 is greater than the pair p3.
The pair p1 is not greater than the pair p4.
```

`operator>=`

Tests if the pair object on the left side of the operator is greater than or equal to the pair object on the right side.

C++

```
template <class Type>
    constexpr bool operator>=(const Type& left, const Type& right);

template <class T, class U>
    constexpr bool operator>=(const pair<T, U>& left, const pair<T, U>&
right);
```

Parameters

left
An object of type `pair` on the left side of the operator.

right
An object of type `pair` on the right side of the operator.

Return Value

`true` if the `pair` on the left side of the operator is greater than or equal to the `pair` on the right side of the operator; otherwise **`false`**.

Remarks

In a comparison of pairs, the values' first elements of the two pairs have the highest priority. If they differ, then the result of their comparison is taken as the result of the comparison of the pair. If the values of the first elements are not different, then the values of the second elements are compared and the result of their comparison is taken as the result of the comparison of the pair.

Example

C++

```cpp
// utility_op_ge.cpp
// compile with: /EHsc
#include <utility>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;
   pair <int, double> p1, p2, p3, p4;

   p1 = make_pair ( 10, 2.22e-1 );
   p2 = make_pair ( 100, 1.11e-1 );
   p3 = make_pair ( 10, 1.11e-1 );
   p4 = make_pair ( 10, 2.22e-1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;
   cout << "The pair p4 is: ( " << p4.first << ", "
        << p4.second << " )." << endl << endl;

   if ( p1 >= p2 )
      cout << "Pair p1 is greater than or equal to pair p2." << endl;
   else
      cout << "The pair p1 is less than the pair p2." << endl;

   if ( p1 >= p3 )
      cout << "Pair p1 is greater than or equal to pair p3." << endl;
   else
      cout << "The pair p1 is less than the pair p3." << endl;

   if ( p1 >= p4 )
      cout << "Pair p1 is greater than or equal to pair p4." << endl;
   else
      cout << "The pair p1 is less than the pair p4." << endl;
}
```

Output

```
The pair p1 is: ( 10, 0.222 ).
The pair p2 is: ( 100, 0.111 ).
The pair p3 is: ( 10, 0.111 ).
The pair p4 is: ( 10, 0.222 ).

The pair p1 is less than the pair p2.
```

```
Pair p1 is greater than or equal to pair p3.
Pair p1 is greater than or equal to pair p4.
```

identity Structure

Article • 08/03/2021

A struct that provides a type definition as the template parameter.

Syntax

```
C++
```

```
struct identity {
   typedef Type type;
   Type operator()(const Type& left) const;
};
```

Parameters

left
The value to identify.

Remarks

The class contains the public type definition `type`, which is the same as the template parameter Type. It is used in conjunction with template function forward to ensure that a function parameter has the desired type.

For compatibility with older code, the class also defines the identity function `operator()` which returns its argument *left*.

`in_place_t`, `in_place_type_t`, `in_place_index_t` struct

Article • 11/15/2024

Introduced in C++17.

The `in_place_t`, `in_place_type_t`, and `in_place_index_t` tag types are used to select the overloaded constructor that creates the object in place in the desired way. For the types that use these tag types, they can also help avoid temporary copy or move operations.

Syntax

C++

```
struct in_place_t
{
    explicit in_place_t() = default;
};

inline constexpr in_place_t in_place{};

template <class T>
struct in_place_type_t
{
    explicit in_place_type_t() = default;
};

template <class T>
constexpr in_place_type_t<T> in_place_type{};

template <size_t I>
struct in_place_index_t
{
    explicit in_place_index_t() = default;
};

template <size_t I>
constexpr in_place_index_t<I> in_place_index{};
```

Parameters

I

The index where the object is created in place.

`T`

The type of object to create.

Remarks

- `in_place_t` indicates in-place construction of an object. Used to create objects in place inside a `std::optional`.
- `in_place_type_t` indicates in-place construction of an object of a specific type. It's useful with `std::any` because `std::any` can hold any kind of type, so we need to specify the type it holds.
- `in_place_index_t` indicates in-place construction of an object at a specific index. Use with `std::variant` to specify the index where the object is created.

The following class types use these structs in their constructors: `expected`, optional, single_view, any or variant.

Example

C++

```
#include <iostream>
#include <optional>
#include <any>
#include <variant>

// compile with /std:c++17

struct MyStruct
{
    double value;
    MyStruct(double v0, double v1 = 0) : value(v0 + v1) {}
};

int main()
{
    // Construct a MyStruct directly inside opt
    std::optional<MyStruct> opt(std::in_place, 29.0, 13.0);

    // Construct a MyStruct object inside an any object
    std::any a(std::in_place_type<MyStruct>, 3.14);

    // Construct a MyStruct object inside a vector at index 0
    std::variant<MyStruct, int> v(std::in_place_index<0>, 2.718);

    if (opt)
    {
        std::cout << opt->value << ", ";
```

```
    }

    std::cout << std::any_cast<MyStruct>(a).value << ", "
              << std::get<0>(v).value;

    return 0;
}
```

Output

```
42, 3.14, 2.718
```

Requirements

Header: `<utility>`

Namespace: `std`

Compiler Option: /std:c++17 or later.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

pair Structure

Article • 08/03/2021

A struct that provides for the ability to treat two objects as a single object.

Syntax

C++

```
struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;
    constexpr pair();
    pair(const pair&) = default;
    pair(pair&&) = default;
    constexpr pair(
        const T1& Val1,
        const T2& Val2);

    template <class Other1, class Other2>
    constexpr pair(const pair<Other1, Other2>& Right);

    template <class Other1, class Other2>
    constexpr pair(const pair <Other1 Val1, Other2 Val2>&& Right);

    template <class Other1, class Other2>
    constexpr pair(Other1&& Val1, Other2&& Val2);

    template <class... Args1, class... Args2>
    pair(piecewise_construct_t, tuple<Args1...> first_args, tuple<Args2...>
second_args);

    pair& operator=(const pair& p);
    template<class U1, class U2> pair& operator=(const pair<U1, U2>& p);
    pair& operator=(pair&& p) noexcept(see below );
    template<class U1, class U2> pair& operator=(pair<U1, U2>&& p);

    void swap(pair& p) noexcept(see below );
};

template<class T1, class T2>
    pair(T1, T2) -> pair<T1, T2>;
```

Parameters

Val1
Value initializing the first element of `pair`.

Val2
Value initializing the second element of `pair`.

Right
A pair whose values are to be used to initialize the elements of another pair.

Return Value

The first (default) constructor initializes the first element of the pair to the default of type `T1` and the second element to default of type `T2`. It is defined if both types are default-constructible.

The second constructor initializes the first element of the pair to *Val1* and the second to *Val2*. It is defined if both types are copy-constructible.

The third (template) constructor initializes the first element of the pair to `Right`. **first** and the second to `Right`. **second**. It is defined if both types of the pair are constructible from the value types provided.

The fourth constructor initializes the first element of the pair to *Val1* and the second to *Val2* using Rvalue Reference Declarator: &&. It is defined if both types of the pair are constructible from the value types provided.

Remarks

The template struct stores a pair of objects of type `T1` and `T2`, respectively. The type `first_type` is the same as the template parameter `T1` and the type `second_type` is the same as the template parameter `T2`. `T1` and `T2` each need supply only a default constructor, a single-argument constructor, and a destructor. All members of the type `pair` are public, because the type is declared as a **`struct`** rather than as a **`class`**. The two most common uses for a pair are as return types for functions that return two values and as elements for the associative container classes map Class and multimap Class that have both a key and a value type associated with each element. The latter satisfies the requirements for a pair associative container and has a value type of the form `pair< const key_type, mapped_type >`.

Example

C++

```
// utility_pair.cpp
// compile with: /EHsc
#include <utility>
#include <map>
#include <iomanip>
#include <iostream>

int main( )
{
   using namespace std;

   // Using the constructor to declare and initialize a pair
   pair <int, double> p1 ( 10, 1.1e-2 );

   // Compare using the helper function to declare and initialize a pair
   pair <int, double> p2;
   p2 = make_pair ( 10, 2.22e-1 );

   // Making a copy of a pair
   pair <int, double> p3 ( p1 );

   cout.precision ( 3 );
   cout << "The pair p1 is: ( " << p1.first << ", "
        << p1.second << " )." << endl;
   cout << "The pair p2 is: ( " << p2.first << ", "
        << p2.second << " )." << endl;
   cout << "The pair p3 is: ( " << p3.first << ", "
        << p3.second << " )." << endl;

   // Using a pair for a map element
   map <int, int> m1;
   map <int, int>::iterator m1_Iter;

   typedef pair <int, int> Map_Int_Pair;

   m1.insert ( Map_Int_Pair ( 1, 10 ) );
   m1.insert ( Map_Int_Pair ( 2, 20 ) );
   m1.insert ( Map_Int_Pair ( 3, 30 ) );

   cout << "The element pairs of the map m1 are:";
   for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )
      cout << " ( " << m1_Iter -> first << ", "
           << m1_Iter -> second << " )";
   cout   << "." << endl;

   // Using pair as a return type for a function
   pair< map<int,int>::iterator, bool > pr1, pr2;
   pr1 = m1.insert ( Map_Int_Pair ( 4, 40 ) );
   pr2 = m1.insert ( Map_Int_Pair (1, 10 ) );

   if( pr1.second == true )
   {
```

```
      cout << "The element (4,40) was inserted successfully in m1."
           << endl;
   }
   else
   {
      cout << "The element with a key value of\n"
           << " ( (pr1.first) -> first ) = " << ( pr1.first ) -> first
           << " is already in m1,\n so the insertion failed." << endl;
   }

   if( pr2.second == true )
   {
      cout << "The element (1,10) was inserted successfully in m1."
           << endl;
   }
   else
   {
      cout << "The element with a key value of\n"
           << " ( (pr2.first) -> first ) = " << ( pr2.first ) -> first
           << " is already in m1,\n so the insertion failed." << endl;
   }
}
```

Output

```
The pair p1 is: ( 10, 0.011 ).
The pair p2 is: ( 10, 0.222 ).
The pair p3 is: ( 10, 0.011 ).
The element pairs of the map m1 are: ( 1, 10 ) ( 2, 20 ) ( 3, 30 ).
The element (4,40) was inserted successfully in m1.
The element with a key value of
( (pr2.first) -> first ) = 1 is already in m1,
so the insertion failed.
```

<valarray>

Article • 05/12/2022

Defines the class template `valarray<Type>` and several supporting class templates and functions.

Requirements

Header: <valarray>

Namespace: std

> ⓘ Note
>
> The <valarray> library uses the `#include <initializer_list>` statement.

Remarks

These class templates and functions are permitted unusual latitude in the interest of improved performance. Specifically, any function that returns type `valarray<T1>` may return an object of some other type `T2`. In that case, any function that accepts one or more arguments of type `valarray<T2>` must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type `T2`.

Members

Functions

Name	Description
abs	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the absolute value of the elements of the input `valarray`.
acos	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the arccosine of the elements of the input `valarray`.
asin	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the arcsine of the elements of the input `valarray`.

Name	Description
atan	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the principal value of the arctangent of the elements of the input `valarray`.
atan2	Returns a `valarray` whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of `valarray` inputs.
begin	
cos	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the cosine of the elements of the input `valarray`.
cosh	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the hyperbolic cosine of the elements of the input `valarray`.
end	
exp	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the natural exponential of the elements of the input `valarray`.
log	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the natural logarithm of the elements of the input `valarray`.
log10	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the base 10 or common logarithm of the elements of the input `valarray`.
pow	Operates on the elements of any input `valarray` and constants, returning a `valarray` whose elements are equal to a base specified either by the elements of an input `valarray` or a constant raised to an exponent specified either by the elements of an input `valarray` or a constant.
sin	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the sine of the elements of the input `valarray`.
sinh	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the hyperbolic sine of the elements of the input `valarray`.
sqrt	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the square root of the elements of the input `valarray`.
swap	
tan	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the tangent of the elements of the input `valarray`.
tanh	Operates on the elements of an input `valarray`, returning a `valarray` whose elements are equal to the hyperbolic tangent of the elements of the input `valarray`.

Operators

Name	Description
operator!=	Tests whether the corresponding elements of two equally sized `valarray` objects are unequal or whether all the elements of a `valarray` are unequal to a specified value of the `valarray` object's element type.
operator%	Obtains the remainder of dividing the corresponding elements of two equally sized `valarray` objects or of dividing a `valarray` by a specified value of the `valarray` object's element type or of dividing a specified value by a `valarray`.
operator&	Obtains the bitwise AND between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the element type.
operator&&	Obtains the logical AND between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the `valarray` object's element type.
operator>	Tests whether the elements of one `valarray` are greater than the elements of an equally sized `valarray` or whether all the elements of a `valarray` are greater or less than a specified value of the `valarray` object's element type.
operator>=	Tests whether the elements of one `valarray` are greater than or equal to the elements of an equally sized `valarray` or whether all the elements of a `valarray` are greater than or equal to or less than or equal to a specified value.
operator>>	Right-shifts the bits for each element of a `valarray` a specified number of positions or by an element-wise amount specified by a second `valarray`.
operator<	Tests whether the elements of one `valarray` are less than the elements of an equally sized `valarray` or whether all the elements of a `valarray` are greater or less than a specified value.
operator<=	Tests whether the elements of one `valarray` are less than or equal to the elements of an equally sized `valarray` or whether all the elements of a `valarray` are greater than or equal to or less than or equal to a specified value.
operator<<	Left shifts the bits for each element of a `valarray` a specified number of positions or by an element-wise amount specified by a second `valarray`.
operator*	Obtains the element-wise product between corresponding elements of two equally sized `valarray` objects or of between a `valarray` a specified value of the `valarray` object's element type.
operator+	Obtains the element-wise sum between corresponding elements of two equally sized `valarray` objects or of between a `valarray` a specified value of the `valarray` object's element type.

Name	Description
operator-	Obtains the element-wise difference between corresponding elements of two equally sized `valarray` objects or of between a `valarray` a specified value of the `valarray` object's element type.
operator/	Obtains the element-wise quotient between corresponding elements of two equally sized `valarray` objects or of between a `valarray` a specified value of the `valarray` object's element type.
operator==	Tests whether the corresponding elements of two equally sized `valarray` objects are equal or whether all the elements of a `valarray` are equal a specified value of the `valarray` object's element type.
operator^	Obtains the bitwise exclusive OR between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the element type.
operator|	Obtains the bitwise OR between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the element type.
operator||	Obtains the logical OR between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the `valarray` object's element type.

Classes

Name	Description
gslice class	A utility class to `valarray` that is used to define multi-dimensional slices of a `valarray`.
gslice_array class	An internal, auxiliary class template that supports general slice objects by providing operations between subset arrays defined by the general slice of a `valarray`.
indirect_array class	An internal, auxiliary class template that supports objects that are subsets of `valarray` objects by providing operations between subset arrays defined by specifying a subset of indices of a parent `valarray`.
mask_array class	An internal, auxiliary class template that supports objects that are subsets of parent `valarray` objects, specified with a Boolean expression, by providing operations between the subset arrays.
slice class	A utility class to `valarray` that is used to define one-dimensional, vector-like subsets of a `valarray`.

Name	Description
slice_array class	An internal, auxiliary class template that supports `slice` objects by providing operations between subset arrays defined by the slice of a `valarray`.
valarray<Type> class	The class template describes an object that controls a sequence of elements of type `Type` that are stored as an array and designed for high-speed mathematical operations, optimized for computational performance.

Specializations

Name	Description
valarray<bool> class	A specialized version of the class template `valarray<Type>` for elements of type `bool`.

See also

Header files reference

Thread safety in the C++ Standard Library

`<valarray>` functions

Article • 12/07/2021

abs

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the absolute value of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> abs(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

C++

```
// valarray_abs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va1 ( 9 ), va2 ( 9 );
   for ( i = 0 ; i < 4 ; i++ )
      va1 [ i ] =  -i;
   for ( i = 4 ; i < 9 ; i++ )
      va1 [ i ] =  i;
```

```
   cout << "The initial valarray is: ";
      for (i = 0 ; i < 9 ; i++ )
         cout << va1 [ i ] << " ";
   cout << "." << endl;

   va2 = abs ( va1 );
   cout << "The absolute value of the initial valarray is: ";
      for (i = 0 ; i < 9 ; i++ )
         cout << va2 [ i ] << " ";
   cout << "." << endl;
}
```

Output

```
The initial valarray is: 0 -1 -2 -3 4 5 6 7 8 .
The absolute value of the initial valarray is: 0 1 2 3 4 5 6 7 8 .
```

acos

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arccosine of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> acos(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arccosine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between 0 and +pi that is consistent with the cosine value input.

Example

C++

```
// valarray_acos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  0.25 * i - 1;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:";
   for (i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = acos ( va1 );
   cout << "The arccosine of the initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
         << "  radians, which is  "
         << setw(11) << (180/pi) * va2 [ i ]
         << "  degrees" << endl;
}
```

Output

```
The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arccosine of the initial valarray is:
   3.14159  radians, which is          180  degrees
   2.41886  radians, which is       138.59  degrees
    2.0944  radians, which is          120  degrees
   1.82348  radians, which is      104.478  degrees
    1.5708  radians, which is           90  degrees
   1.31812  radians, which is      75.5225  degrees
    1.0472  radians, which is           60  degrees
  0.722734  radians, which is      41.4096  degrees
         0  radians, which is            0  degrees
```

asin

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the arcsine of the elements of the input valarray.

```
C++
```

```
template <class Type>
valarray<Type> asin(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arcsine of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between +pi/2 and -pi/2 that is consistent with the sine value input.

Example

```
C++
```

```
// valarray_asin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  0.25 * i - 1;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:";
```

```
   for (i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = asin ( va1 );
   cout << "The arcsine of the initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
           << "  radians, which is  "
           << setw(11) << (180/pi) * va2 [ i ]
           << "  degrees" << endl;
}
```

Output

```
The initial valarray is: -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1
The arcsine of the initial valarray is:
   -1.5708  radians, which is          -90  degrees
 -0.848062  radians, which is     -48.5904  degrees
 -0.523599  radians, which is          -30  degrees
  -0.25268  radians, which is     -14.4775  degrees
         0  radians, which is            0  degrees
   0.25268  radians, which is      14.4775  degrees
  0.523599  radians, which is           30  degrees
  0.848062  radians, which is      48.5904  degrees
    1.5708  radians, which is           90  degrees
```

atan

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the principal value of the arctangent of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> atan(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the arctangent of the elements of the input valarray.

Remarks

The units of the returned elements are in radians.

The return value is a principal value between +pi/2 and -pi/2 that is consistent with the tangent value input.

Example

C++

```
// valarray_atan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   va1 [ 0 ] = -100;
   for ( i = 1 ; i < 8 ; i++ )
      va1 [ i ] =  5 * ( 0.25 * i - 1 );
   va1 [ 8 ] = 100;
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is: ";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va1 [ i ] << " ";
   cout << "." << endl;

   va2 = atan ( va1 );
   cout << "The arcsine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va2 [ i ]
           << "  radians, which is  "
           << setw(11) << (180/pi) * va2 [ i ]
           << "  degrees" << endl;
   cout << endl;
}
```

Output

```
The initial valarray is: -100 -3.75 -2.5 -1.25 0 1.25 2.5 3.75 100 .
The arcsine of the initial valarray is:
   -1.5608  radians, which is     -89.4271  degrees
  -1.31019  radians, which is     -75.0686  degrees
  -1.19029  radians, which is     -68.1986  degrees
 -0.896055  radians, which is     -51.3402  degrees
         0  radians, which is            0  degrees
  0.896055  radians, which is      51.3402  degrees
   1.19029  radians, which is      68.1986  degrees
   1.31019  radians, which is      75.0686  degrees
    1.5608  radians, which is      89.4271  degrees
```

atan2

Returns a valarray whose elements are equal to the arctangent of the Cartesian components specified by a combination of constants and elements of valarrays.

C++

```
template <class Type>
valarray<Type> atan2(const valarray<Type>& left, const valarray<Type>&
right);

template <class Type>
valarray<Type> atan2(const valarray<Type> left, const Type& right);

template <class Type>
valarray<Type> atan2(const Type& left, const valarray<Type>& right);
```

Parameters

left
The constant numerical data type or input valarray whose elements provide the values for the y-coordinate of the arctangent argument.

right
The constant numerical data type or input valarray whose elements provide the values for the x-coordinate of the arctangent argument.

Return Value

A valarray whose elements `I` are equal to the arctangent of:

- `left` [*I*] / *_Right* [*I*] for the first template function.

- `left` [/] / `right` for the second template function.
- `left` / `right` [/] for the third template function.

Remarks

The units of the returned elements are in radians.

This function preserves information about the signs of the components in the argument that is lost by the standard tangent function, and this knowledge of the quadrant enables the return value to be assigned a unique angle between +pi and -pi.

If *left* and *right* have a different number of elements, the result is undefined.

Example

C++

```
// valarray_atan2.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1y ( 1 , 4 ), va1x ( 1 , 4 );
   va1x [ 1 ] = -1;
   va1x [ 2 ] = -1;
   va1y [ 2 ] = -1;
   va1y [ 3 ] = -1;
   valarray<double> va2 ( 4 );

   cout << "The initial valarray for the x coordinate is: ( ";
   for ( i = 0 ; i < 4 ; i++ )
      cout << va1x [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial valarray for the y coordinate is: ( ";
   for ( i = 0 ; i < 4 ; i++ )
      cout << va1y [ i ] << " ";
   cout << ")." << endl;

   va2 = atan2 ( va1y , va1x );
   cout << "The atan2 ( y / x ) of the initial valarrays is:\n";
   for ( i = 0 ; i < 4 ; i++ )
```

```
      cout << setw( 10 ) << va2 [ i ]
           << "  radians, which is  "
           << setw( 11 ) << ( 180/pi ) * va2 [ i ]
           << "degrees" << endl;
   cout << endl;
}
```

Output

```
The initial valarray for the x coordinate is: ( 1 -1 -1 1 ).
The initial valarray for the y coordinate is: ( 1 1 -1 -1 ).
The atan2 ( y / x ) of the initial valarrays is:
  0.785398  radians, which is           45degrees
   2.35619  radians, which is          135degrees
  -2.35619  radians, which is         -135degrees
 -0.785398  radians, which is          -45degrees
```

begin

C++

```
template <class T> unspecified 1 begin(valarray<T>& v);
template <class T> unspecified 2 begin(const valarray<T>& v);
```

cos

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the cosine of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> cos(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

C++

```
// valarray_cos.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  ( pi ) * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "  radians, which is  "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = cos ( va1 );
   cout << "The cosine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159  radians, which is   -180  degrees
  -2.35619  radians, which is   -135  degrees
   -1.5708  radians, which is    -90  degrees
 -0.785398  radians, which is    -45  degrees
         0  radians, which is      0  degrees
  0.785398  radians, which is     45  degrees
    1.5708  radians, which is     90  degrees
   2.35619  radians, which is    135  degrees
   3.14159  radians, which is    180  degrees
```

```
The cosine of the initial valarray is:
-1
-0.707107
-1.03412e-013
0.707107
1
0.707107
-1.03412e-013
-0.707107
-1
```

cosh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> cosh(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic cosine in terms of exponential function:

cosh (*z*) = (exp (*z*) + exp (- *z*)) / 2

Example

C++

```
// valarray_cosh.cpp
// compile with: /EHsc
```

```
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "  radians, which is  "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = cosh ( va1 );
   cout << "The hyperbolic cosine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159  radians, which is   -180  degrees
  -2.35619  radians, which is   -135  degrees
   -1.5708  radians, which is    -90  degrees
 -0.785398  radians, which is    -45  degrees
         0  radians, which is      0  degrees
  0.785398  radians, which is     45  degrees
    1.5708  radians, which is     90  degrees
   2.35619  radians, which is    135  degrees
   3.14159  radians, which is    180  degrees

The hyperbolic cosine of the initial valarray is:
11.592
5.32275
2.50918
1.32461
1
1.32461
2.50918
5.32275
11.592
```

end

```
C++
```

```
template <class T> unspecified 1 end(valarray<T>& v);
template <class T> unspecified 2 end(const valarray<T>& v);
```

exp

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural exponential of the elements of the input valarray.

```
C++
```

```
template <class Type>
valarray<Type> exp(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the natural exponential of the elements of the input valarray.

Example

```
C++
```

```
// valarray_exp.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 9 );
```

```
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  10 * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for ( i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = exp ( va1 );
   cout << "The natural exponential of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: -10 -7.5 -5 -2.5 0 2.5 5 7.5 10
The natural exponential of the initial valarray is:
4.53999e-005
0.000553084
0.00673795
0.082085
1
12.1825
148.413
1808.04
22026.5
```

log

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the natural logarithm of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> log(const valarray<Type>& left);
```

Parameters

left

The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the absolute value of the elements of the input valarray.

Example

C++

```
// valarray_log.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 9 );
   for (i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  10 * i;
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for ( i = 0 ; i < 9 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = log ( va1 );
   cout << "The natural logarithm of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: 0 10 20 30 40 50 60 70 80
The natural logarithm of the initial valarray is:
-inf
2.30259
2.99573
3.4012
3.68888
3.91202
4.09434
4.2485
4.38203
```

log10

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the base 10 or common logarithm of the elements of the input valarray.

```
C++
```

```
template <class Type>
valarray<Type> log10(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the common logarithm of the elements of the input valarray.

Example

```
C++
```

```
// valarray_log10.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 11 );
   for ( i = 0 ; i < 11 ; i++ )
      va1 [ i ] =  10 * i;
   valarray<double> va2 ( 9 );

   cout << "Initial valarray:";
   for (i = 0 ; i < 11 ; i++ )
      cout << " " << va1 [ i ];
   cout << endl;

   va2 = log10 ( va1 );
```

```
   cout << "The common logarithm of the initial valarray is:\n";
   for (i = 0 ; i < 11 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
Initial valarray: 0 10 20 30 40 50 60 70 80 90 100
The common logarithm of the initial valarray is:
-inf
1
1.30103
1.47712
1.60206
1.69897
1.77815
1.8451
1.90309
1.95424
2
```

pow

Operates on the elements of input valarrays and constants, returning a valarray whose elements are equal to a base specified either by the elements of an input valarray or a constant raised to an exponent specified either by the elements of an input valarray or a constant.

C++

```
template <class Type>
valarray<Type>
pow(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
pow(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
pow(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left
The input valarray whose elements supply the base for each element to be exponentiated.

right
The input valarray whose elements supply the power for each element to be exponentiated.

Return Value

A valarray whose elements `I` are equal to:

- `left` [*I*] raised to the power `right` [*I*] for the first template function.
- `left` [*I*] raised to the power `right` for the second template function.
- `left` raised to the power `right` [*I*] for the third template function.

Remarks

If *left* and *right* have a different number of elements, the result is undefined.

Example

C++

```
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> vabase ( 6 );
   for ( i = 0 ; i < 6 ; i++ )
      vabase [ i ] =  i/2;
   valarray<double> vaexp ( 6 );
   for ( i = 0 ; i < 6 ; i++ )
      vaexp [ i ] =  2 * i;

   valarray<double> va2 ( 6 );
```

```
   cout << "The initial valarray for the base is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vabase [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial valarray for the exponent is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaexp[ i ] << " ";
   cout << ")." << endl;

   va2 = pow ( vabase , vaexp );
   cout << "The power of (n/2) * exp (2n) for n = 0 to n = 5 is: \n";
      for ( i = 0 ; i < 6 ; i++ )
         cout <<  "n = " << i << "\tgives " << va2 [ i ] << endl;
}
```

Output

```
The initial valarray for the base is: ( 0 0 1 1 2 2 ).
The initial valarray for the exponent is: ( 0 2 4 6 8 10 ).
The power of (n/2) * exp (2n) for n = 0 to n = 5 is:
n = 0   gives 1
n = 1   gives 0
n = 2   gives 1
n = 3   gives 1
n = 4   gives 256
n = 5   gives 1024
```

sin

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the sine of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> sin(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the sine of the elements of the input valarray.

Example

C++

```
// valarray_sin.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw(10) << va1 [ i ]
      << "   radians, which is   "
      << setw(5) << ( 180/pi ) * va1 [ i ]
      << "   degrees" << endl;
   cout << endl;

   va2 = sin ( va1 );
   cout << "The sine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is    -180   degrees
  -2.35619   radians, which is    -135   degrees
   -1.5708   radians, which is     -90   degrees
 -0.785398   radians, which is     -45   degrees
         0   radians, which is       0   degrees
  0.785398   radians, which is      45   degrees
    1.5708   radians, which is      90   degrees
   2.35619   radians, which is     135   degrees
   3.14159   radians, which is     180   degrees

The sine of the initial valarray is:
2.06823e-013
-0.707107
-1
```

```
-0.707107
0
0.707107
1
0.707107
-2.06823e-013
```

sinh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> sinh(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic sine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic sine in terms of exponential function:

sinh (*z*) = (exp (*z*) - exp (- *z*)) / 2

Example

C++

```
// valarray_sinh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>
```

```
int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for (i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for (i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "   radians, which is   "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "  degrees" << endl;
   cout << endl;

   va2 = sinh ( va1 );
   cout << "The hyperbolic sine of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is    -180  degrees
  -2.35619   radians, which is    -135  degrees
   -1.5708   radians, which is     -90  degrees
 -0.785398   radians, which is     -45  degrees
         0   radians, which is       0  degrees
  0.785398   radians, which is      45  degrees
    1.5708   radians, which is      90  degrees
   2.35619   radians, which is     135  degrees
   3.14159   radians, which is     180  degrees

The hyperbolic sine of the initial valarray is:
-11.5487
-5.22797
-2.3013
-0.868671
0
0.868671
2.3013
5.22797
11.5487
```

sqrt

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the square root of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> sqrt(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the square root of the elements of the input valarray.

Example

C++

```
// valarray_sqrt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <cmath>

int main( )
{
   using namespace std;
   int i;

   valarray<double> va1 ( 6 );
   for ( i = 0 ; i < 5 ; i++ )
      va1 [ i ] = i * i;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << va1 [ i ] << " ";
   cout << ")." << endl;

   valarray<double> va2 = sqrt ( va1 );
   cout << "The square root of the initial valarray is: ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << va2 [ i ] << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 0 1 4 9 16 ).
The square root of the initial valarray is: ( 0 1 2 3 4 ).
```

swap

Exchanges the elements of two valarrays.

C++

```
template <class Type>
void swap(
    valarray<Type>& left,
    valarray<Type>& right);
```

Parameters

left
An object of type `valarray`.

right
An object of type `valarray`.

Remarks

The template function executes `left.swap(right)`.

tan

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the tangent of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> tan(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the tangent of the elements of the input valarray.

Example

C++

```
// valarray_tan.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  ( pi/2 ) * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "     radians, which is   "
      << setw( 5 ) << ( 180/pi ) * va1 [ i ]
      << "   degrees" << endl;
   cout << endl;

   va2 = tan ( va1 );
   cout << "The tangent of the initial valarray is:\n ";
   for ( i = 0 ; i < 9 ; i++ )
      cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
   -1.5708     radians, which is     -90   degrees
```

```
   -1.1781     radians, which is   -67.5   degrees
 -0.785398     radians, which is     -45   degrees
 -0.392699     radians, which is   -22.5   degrees
         0     radians, which is       0   degrees
  0.392699     radians, which is    22.5   degrees
  0.785398     radians, which is      45   degrees
    1.1781     radians, which is    67.5   degrees
    1.5708     radians, which is      90   degrees

The tangent of the initial valarray is:
 9.6701e+012
-2.41421
-1
-0.414214
0
0.414214
1
2.41421
-9.6701e+012
```

tanh

Operates on the elements of an input valarray, returning a valarray whose elements are equal to the hyperbolic tangent of the elements of the input valarray.

C++

```
template <class Type>
valarray<Type> tanh(const valarray<Type>& left);
```

Parameters

left
The input valarray whose elements are to be operated on by the member function.

Return Value

A valarray whose elements are equal to the hyperbolic cosine of the elements of the input valarray.

Remarks

Identities defining the hyperbolic tangent in terms of the exponential function:

tanh (*z*) = sinh (*z*) / cosh (*z*) = (exp (*z*) - exp (- *z*)) / (exp (*z*) + exp (- *z*))

Example

C++

```
// valarray_tanh.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
#include <iomanip>

int main( )
{
   using namespace std;
   double pi = 3.14159265359;
   int i;

   valarray<double> va1 ( 9 );
   for ( i = 0 ; i < 9 ; i++ )
      va1 [ i ] =  pi * ( 0.25 * i - 1 );
   valarray<double> va2 ( 9 );

   cout << "The initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
      cout << setw( 10 ) << va1 [ i ]
      << "   radians, which is     "
                  << setw( 5 ) << ( 180/pi ) * va1 [ i ]
                  << "  degrees" << endl;
   cout << endl;

   va2 = tanh ( va1 );
   cout << "The hyperbolic tangent of the initial valarray is:\n";
   for ( i = 0 ; i < 9 ; i++ )
     cout << va2 [ i ] << endl;
}
```

Output

```
The initial valarray is:
  -3.14159   radians, which is      -180  degrees
  -2.35619   radians, which is      -135  degrees
   -1.5708   radians, which is       -90  degrees
 -0.785398   radians, which is       -45  degrees
         0   radians, which is         0  degrees
  0.785398   radians, which is        45  degrees
    1.5708   radians, which is        90  degrees
   2.35619   radians, which is       135  degrees
   3.14159   radians, which is       180  degrees

The hyperbolic tangent of the initial valarray is:
-0.996272
-0.982193
-0.917152
```

```
-0.655794
0
0.655794
0.917152
0.982193
0.996272
```

`<valarray>` operators

Article • 05/17/2022

`operator!=`

Tests whether the corresponding elements of two equally sized `valarray` objects are unequal, or whether all the elements of a `valarray` are unequal to a specified value.

C++

```
template <class Type>
valarray<bool>
operator!=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator!=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator!=(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

`left`

A `valarray` object whose elements are to be tested for inequality against another `valarray` object, or a value of element type to compare against each element in a `valarray`.

`right`

A `valarray` object whose elements are to be tested for inequality against another `valarray` object, or a value of element type to compare against each element in a `valarray`.

Return value

A `valarray` of Boolean values, each of which is:

- `true` if the corresponding elements are unequal.
- `false` if the corresponding elements are equal.

Remarks

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `left[I] != right[I]`.

The second template operator stores `left[I] != right` in element `I`.

The third template operator stores `left != right[I]` in element `I`.

Example

C++

```
// valarray_op_ne.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL != vaR );
```

```
   cout << "The element-by-element result of "
        << "the not equal comparison test is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the not equal comparison test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

operator%

Obtains the remainder of dividing the corresponding elements of two equally sized `valarray` objects, or of dividing a `valarray` by a specified value, or of dividing a specified value by a `valarray`.

C++

```
template <class Type>
valarray<Type>
operator%(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator%(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator%(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

`left`

A value or `valarray` that serves as the dividend into which another value or `valarray` is to be divided.

right

A value or valarray that serves as the divisor and that divides another value or valarray.

Return value

A valarray whose elements are the element-wise remainders of *left* divided by *right*.

Example

C++

```
// valarray_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 6 ), vaR ( 6 );
   valarray<int> vaREM ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  53;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -67;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  3*i+1;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaREM = ( vaL % vaR );
   cout << "The remainders from the element-by-element "
        << "division is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaREM [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 53 -67 53 -67 53 -67 ).
The initial Right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
valarray: ( 0 -3 4 -7 1 -3 ).
```

`operator&`

Obtains the bitwise AND between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value of the element type.

C++

```
template <class Type>
valarray<Type>
operator&(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator&(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator&(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

`left`

The first of the two `valarray` objects whose respective elements are to be combined with the bitwise AND, or a specified value of the element type to be combined by bitwise AND with each element of a `valarray`.

`right`

The second of the two `valarray` objects whose respective elements are to be combined with the bitwise AND, or a specified value of the element type to be combined by bitwise AND with each element of a `valarray`.

Return Value

A `valarray` whose elements are the element-wise combination of the bitwise AND operation of *`left`* and *`right`*.

Remarks

A bitwise operation can only be used to manipulate bits in **`char`** and **`int`** data types and variants and not on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`** or other, more complex data types.

The bitwise AND has the same truth table as the logical AND but applies to the data type on the level of the individual bits. The operator&& applies on an element level, counting all nonzero values as `true`, and the result is a `valarray` of Boolean values. The bitwise AND operator&, by contrast, can result in a `valarray` of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

C++

```
// valarray_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaBWA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i+1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
```

```
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaBWA = ( vaL & vaR );
   cout << "The element-by-element result of "
        << "the bitwise operator & is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaBWA [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is:  ( 0 2 0 4 0 6 0 8 0 10 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the bitwise operator & is the
valarray: ( 0 0 0 0 0 4 0 0 0 8 ).
```

operator&&

Obtains the logical AND between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value of the `valarray` element type.

C++

```
template <class Type>
valarray<bool>
operator&&(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator&&(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator&&(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left
The first of the two `valarray` objects whose respective elements are to be combined with the logical AND, or a specified value of the element type to be combined with each element of a `valarray`.

right
The second of the two `valarray` objects whose respective elements are to be combined with the logical AND, or a specified value of the element type to be combined with each element of a `valarray`.

Return value

A `valarray` whose elements are of type `bool` and are the element-wise combination of the logical AND operation of *`left`* and *`right`*.

Remarks

The logical AND `operator&&` applies on an element level, counting all nonzero values as `true`, and the result is a `valarray` of Boolean values. The bitwise version of AND, operator&, by contrast, can result in a `valarray` of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

C++

```
// valarray_op_logand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;
```

```
   cout << "The initial Left valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaLAA = ( vaL && vaR );
   cout << "The element-by-element result of "
        << "the logical AND operator&& is the\n"
        << "valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaLAA [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the logical AND operator&& is the
valarray: ( 0 0 0 1 0 1 0 1 0 1 ).
```

`operator>`

Tests whether the elements of one `valarray` are greater than the elements of an equally sized `valarray`, or whether all the elements of a `valarray` are greater or less than a specified value.

C++

```
template <class Type>
valarray<bool>
operator>(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator>(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator>(
```

```
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

Return value

A `valarray` of Boolean values, each of which is:

- **`true`** if the *`left`* element or value is greater than the corresponding *`right`* element or value.
- **`false`** if the *`left`* element or value isn't greater than the corresponding *`right`* element or value.

Remarks

If the number of elements in the two `valarray` objects isn't equal, the result is undefined.

Example

C++

```
// valarray_op_gt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
```

```
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL > vaR );
   cout << "The element-by-element result of "
        << "the greater than comparison test is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than comparison test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

`operator>=`

Tests whether the elements of one `valarray` are greater than or equal to the elements of an equally sized `valarray`, or whether all the elements of a `valarray` are greater than or equal to or less than or equal to a specified value.

C++

```
template <class Type>
valarray<bool>
operator>=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
```

```
operator>=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator>=(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

Return value

A `valarray` of Boolean values, each of which is:

- **`true`** if the *`left`* element or value is greater than or equal to the corresponding *`right`* element or value.
- **`false`** if the *`left`* element or value is less than the corresponding *`right`* element or value.

Remarks

If the number of elements in two `valarray` objects isn't equal, the result is undefined.

Example

C++

```
// valarray_op_ge.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL >= vaR );
   cout << "The element-by-element result of "
        << "the greater than or equal test is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the greater than or equal test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

operator>>

Right-shifts the bits for each element of a `valarray` a specified number of positions, or by an element-wise amount specified by a second `valarray`.

C++

```
template <class Type>
valarray<Type>
operator>>(
    const valarray<Type>& left,
```

```
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator>>(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator>>(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The value to be shifted, or the `valarray` whose elements are to be shifted.

right

The value indicating the amount of right shift, or a `valarray` whose elements indicate the element-wise amount of right shift.

Return value

A `valarray` whose elements have been shifted right by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

C++

```
// valarray_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
```

```
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  64;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -64;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL >> vaR );
   cout << "The element-by-element result of "
        << "the right shift is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
```

operator<

Tests whether the elements of one `valarray` are less than the elements of an equally sized `valarray`, or whether all the elements of a `valarray` are greater or less than a specified value.

C++

```
template <class Type>
valarray<bool>
operator<(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
```

```
valarray<bool>
operator<(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator<(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

Return value

A `valarray` of Boolean values, each of which is:

- **`true`** if the *`left`* element or value is less than the corresponding *`right`* element or value.
- **`false`** if the *`left`* element or value isn't less than the corresponding *`right`* element or value.

Remarks

If the number of elements in two `valarray` objects isn't equal, the result is undefined.

Example

C++

```
// valarray_op_lt.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL < vaR );
   cout << "The element-by-element result of "
        << "the less-than comparson test is the\n"
        << "valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the less-than comparson test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

operator<=

Tests whether the elements of one `valarray` are less than or equal to the elements of an equally sized `valarray`, or whether all the elements of a `valarray` are greater than or equal to or less than or equal to a specified value.

C++

```
template <class Type>
valarray<bool>
operator<=(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator<=(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator<=(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be compared, or a specified value to be compared with each element of a `valarray`.

Return value

A `valarray` of Boolean values, each of which is:

- **`true`** if the *`left`* element or value is less than or equal to the corresponding *`right`* element or value.
- **`false`** if the *`left`* element or value is greater than the corresponding *`right`* element or value.

Remarks

If the number of elements in two `valarray` objects isn't equal, the result is undefined.

Example

C++

```
// valarray_op_le.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i - 1;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL <= vaR );
   cout << "The element-by-element result of "
        << "the less than or equal test is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( -1 0 1 2 3 4 5 6 7 8 ).
The element-by-element result of the less than or equal test is the
valarray: ( 0 0 1 0 1 0 1 0 1 0 ).
```

operator<<

Left shifts the bits for each element of a `valarray` a specified number of positions, or by an element-wise amount specified by a second `valarray`.

```
C++
```

```
template <class Type>
valarray<Type>
operator<<(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator<<(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator<<(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

`left`

The value to be shifted, or the `valarray` whose elements are to be shifted.

`right`

The value indicating the amount of left shift, or a `valarray` whose elements indicate the element-wise amount of left shift.

Return value

A `valarray` whose elements have been shifted left by the specified amount.

Remarks

Signed numbers have their signs preserved.

Example

```
C++
```

```cpp
// valarray_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL << vaR );
   cout << "The element-by-element result of "
        << "the left shift is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift is the
valarray: ( 1 -2 4 -8 16 -32 64 -128 ).
```

`operator*`

Obtains the element-wise product between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value.

```
C++
```

```
template <class Type>
valarray<Type>
operator*(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator*(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator*(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be multiplied, or a specified value to be multiplied with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be multiplied, or a specified value to be multiplied with each element of a `valarray`.

Return value

A `valarray` whose elements are the element-wise product of *left* and *right*.

Example

```
C++
```

```
// valarray_op_eprod.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL * vaR );
   cout << "The element-by-element result of "
        << "the multiplication is the\n"
        << "valarray: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
```

operator+

Obtains the element-wise sum between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value.

C++

```
template <class Type>
valarray<Type>
operator+(
    const valarray<Type>& left,
    const valarray<Type>& right);
```

```
template <class Type>
valarray<Type>
operator+(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator+(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be added, or a specified value to be added with each element of a `valarray`.

right

The second of the two `valarray` objects whose elements are to be added, or a specified value to be added with each element of a `valarray`.

Return value

A `valarray` whose elements are the element-wise sum of *left* and *right*.

Example

C++

```
// valarray_op_esum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
```

```
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL + vaR );
   cout << "The element-by-element result of "
        << "the sum is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the sum is the
valarray: ( 2 0 4 2 6 4 8 6 ).
```

operator-

Obtains the element-wise difference between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value.

C++

```
template <class Type>
valarray<Type>
operator-(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator-(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator-(
```

```
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

A value or `valarray` that serves as the *minuend*, the value from which other values or `valarray` objects are to be subtracted in forming the difference.

right

A value or `valarray` that serves as the *subtrahend*, the value that is to be subtracted from other values or `valarray` objects in forming the difference.

Return value

A `valarray` whose elements are the element-wise difference of *left* and *right*.

Remarks

The arithmetic terminology used in describing a subtraction:

difference = minuend - subtrahend

Example

C++

```
// valarray_op_ediff.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   valarray<int> vaNE ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  10;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;
```

```
   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL - vaR );
   cout << "The element-by-element result of "
        << "the difference is the\n"
        << "valarray: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
```

`operator/`

Obtains the element-wise quotient between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value.

C++

```
template <class Type>
valarray<Type>
operator/(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator/(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator/(
```

```
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

A value or `valarray` that serves as the *dividend* into which another value or `valarray` is to be divided in forming the quotient.

right

A value or `valarray` that serves as the *divisor*, and that divides another value or `valarray` in forming the quotient.

Return value

A `valarray` whose elements are the element-wise quotient of *left* divided by *right*.

Remarks

The arithmetic terminology used in describing a division:

quotient = dividend / divisor

Example

C++

```
// valarray_op_equo.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<double> vaL ( 6 ), vaR ( 6 );
   valarray<double> vaNE ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  100;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -100;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  2*i;
```

```
   cout << "The initial Left valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL / vaR );
   cout << "The element-by-element result of "
        << "the quotient is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
valarray: ( inf -50 25 -16.6667 12.5 -10 ).
```

`operator==`

Tests whether the corresponding elements of two equally sized `valarray` objects are equal, or whether all the elements of a `valarray` are equal to a specified value.

C++

```
template <class Type>
valarray<bool>
operator==(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator==(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator==(
```

```
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose elements are to be tested for equality, or a specified value to compare with each `valarray` element.

right

The second of the two `valarray` objects whose elements are to be tested for equality, or a specified value to compare with each `valarray` element.

Return Value

A `valarray` of Boolean values, each of which is:

- **`true`** if the corresponding elements are equal.
- **`false`** if the corresponding elements aren't equal.

Remarks

The first template operator returns an object of class valarray<bool>, each of whose elements `I` is `left[I] == right[I]`. The second template operator stores `left[I] == right` in element `I`. The third template operator stores `left == right[I]` in element `I`.

Example

C++

```
// valarray_op_eq.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaNE ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
```

```
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaNE = ( vaL == vaR );
   cout << "The element-by-element result of "
        << "the equality comparison test is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNE [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the equality comparison test is the
valarray: ( 1 1 0 1 0 1 0 1 0 1 ).
```

operator^

Obtains the bitwise exclusive OR (XOR) between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value of the element type.

C++

```
template <class Type>
valarray<Type>
operator^(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator^(
```

```
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator^(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose respective elements are to be combined with the bitwise XOR, or a specified value of the element type to be combined by a bitwise XOR with each element of a `valarray`.

right

The second of the two `valarray` objects whose respective elements are to be combined with the bitwise XOR, or a specified value of the element type to be combined by a bitwise XOR with each element of a `valarray`.

Return value

A `valarray` whose elements are the element-wise combination of the bitwise XOR operation of *left* and *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **`char`** and **`int`** data types and variants and not on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`** or other, more complex data types.

The bitwise exclusive OR (XOR) has the following semantics: Given bits *b*1 and *b*2, *b*1 XOR *b*2 is 1 if exactly one of the bits is 1; 0 if both bits are 0 or if both bits are 1.

Example

C++

```
// valarray_op_xor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;

   cout << "The initial Left valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaLAA = ( vaL ^ vaR );
   cout << "The element-by-element result of "
        << "the bitwise XOR operator^ is the\n"
        << "valarray: ( ";
          for ( i = 0 ; i < 10 ; i++ )
         cout << vaLAA [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^ is the
valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
```

`operator|`

Obtains the bitwise OR between corresponding elements of two equally sized `valarray` objects or between a `valarray` and a specified value of the element type.

C++

```
template <class Type>
valarray<Type>
operator|(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<Type>
operator|(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<Type>
operator|(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

left

The first of the two `valarray` objects whose respective elements are to be combined with the bitwise OR, or a specified value of the element type to be combined by bitwise OR with each element of a `valarray`.

right

The second of the two `valarray` objects whose respective elements are to be combined with the bitwise OR, or a specified value of the element type to be combined by bitwise OR with each element of a `valarray`.

Return value

A `valarray` whose elements are the element-wise combination of the bitwise OR operation of *left* and *right*.

Remarks

A bitwise operation can only be used to manipulate bits in **`char`** and **`int`** data types and variants and not on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`** or other, more complex data types.

The bitwise OR has the same truth table as the logical OR, but applies to the data type on the level of the individual bits. Given bits *b*1 and *b*2, *b*1 OR *b*2 is 1 if at least one of the bits is 1, or 0 if both bits are 0. The logical OR operator|| applies on an element level, counting all nonzero values as `true`, and the result is a `valarray` of Boolean values. The bitwise OR `operator|`, by contrast, can result in a `valarray` of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

C++

```
// valarray_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<int> vaLAA ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;

   cout << "The initial Left valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaLAA = ( vaL | vaR );
   cout << "The element-by-element result of "
        << "the bitwise OR operator| is the\n"
        << "valarray: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaLAA [ i ] << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
The initial Left valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The initial Right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise OR operator| is the
valarray: ( 1 0 1 3 3 4 7 6 7 9 ).
```

`operator||`

Obtains the logical OR between corresponding elements of two equally sized `valarray` objects, or between a `valarray` and a specified value of the `valarray` element type.

C++

```
template <class Type>
valarray<bool>
operator||(
    const valarray<Type>& left,
    const valarray<Type>& right);

template <class Type>
valarray<bool>
operator||(
    const valarray<Type>& left,
    const Type& right);

template <class Type>
valarray<bool>
operator||(
    const Type& left,
    const valarray<Type>& right);
```

Parameters

`left`

The first of the two `valarray` objects whose respective elements are to be combined with the logical OR, or a specified value of the element type to be combined with each element of a `valarray`.

`right`

The second of the two `valarray` objects whose respective elements are to be combined

with the logical OR, or a specified value of the element type to be combined with each element of a `valarray`.

Return value

A `valarray` whose elements are of type **`bool`** and are the element-wise combination of the logical OR operation of *`left`* and *`right`*.

Remarks

The logical OR `operator||` applies on an element level, counting all nonzero values as **`true`**, and the result is a `valarray` of Boolean values. The bitwise version of OR, operator|, by contrast, can result in a `valarray` of values other than 0 or 1, depending on the outcome of the bitwise operation.

Example

C++

```
// valarray_op_logor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaLOR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  0;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  0;

   cout << "The initial Left valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
```

```
   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaLOR = ( vaL || vaR );
   cout << "The element-by-element result of "
        << "the logical OR operator|| is the\n"
        << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaLOR [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial Left valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 0 0 3 0 0 6 0 0 9 ).
The element-by-element result of the logical OR operator|| is the
valarray: ( 0 0 0 1 0 1 1 1 0 1 ).
```

`gslice` class

Article • 05/12/2022

A utility class to `valarray` that is used to define multidimensional subsets of a `valarray`. If a `valarray` is regarded as a multidimensional matrix with all elements in an array, then the `gslice` extracts a vector out of the multidimensional array.

Remarks

The class stores the parameters that characterize an object of type gslice_array. The subset of a `valarray` is indirectly constructed when an object of class `gslice` appears as an argument for an object of class valarray<Type>. The stored values that specify the subset selected from the parent `valarray` include:

- A starting index.
- A length vector of class `valarray<size_t>`.
- A stride vector of class `valarray<size_t>`.

The two vectors must have the same length.

If the set defined by a `gslice` is the subset of a constant `valarray`, then the `gslice` is a new `valarray`. If the set defined by a `gslice` is the subset of a nonconstant `valarray`, then the `gslice` has reference semantics to the original `valarray`. The evaluation mechanism for nonconstant `valarray` objects saves time and memory.

Operations on `valarray` objects are guaranteed only if the source and destination subsets defined by the `gslice` objects are distinct and all indices are valid.

Constructors

Constructor	Description
gslice	Defines a subset of a `valarray` that consists of multiple slices of the `valarray` that all start at a specified element.

Member functions

Member function	Description
size	Finds the array values specifying the numbers of elements in a general slice of a `valarray`.
start	Finds the starting index of a general slice of a `valarray`.
stride	Finds the distance between elements in a general slice of a `valarray`.

Requirements

Header: <valarray>

Namespace: std

`gslice::gslice`

A utility class to `valarray` that is used to define multi-dimensional slices of a `valarray`.

```
C++

gslice();

gslice(
    size_t _StartIndex,
    const valarray<size_t>& _LenArray,
    const valarray<size_t>& _IncArray);
```

Parameters

`_StartIndex`

The `valarray` index of the first element in the subset.

`_LenArray`

A `valarray` specifying the number of elements in each `gslice`.

`_IncArray`

A `valarray` specifying the stride in each `gslice`.

Return value

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores `_StartIndex` for the starting index, `_LenArray` for the length array, and `_IncArray` for the stride array.

Remarks

`gslice` defines a subset of a `valarray` that consists of multiple slices of the `valarray` that each start at the same specified element. The ability to use arrays to define multiple slices is the only difference between `gslice` and slice::slice. The first slice has a first element with an index of `_StartIndex`, a count of elements specified by the first element of `_LenArray`, and a stride given by the first element of `_IncArray`. The next set of orthogonal slices has first elements given by the first slice. The second element of `_LenArray` specifies the number of elements. The stride is given by the second element of `_IncArray`. A third dimension of slices would take the elements of the two-dimensional array as the starting elements and continue analogously.

Example

C++

```
// gslice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:" << endl << "(";
   for ( i = 0 ; i < 20 ; i++ )
      cout << " " << va [ i ];
   cout << " )" << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
```

```
   cout << "The valarray for vaGSlice is vaResult:" << endl
        << "va[vaGSlice] = (";

   for ( i = 0 ; i < 8 ; i++ )
      cout << " " << vaResult [ i ];
   cout << ")" << endl;
}
```

Output

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 )
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19)
```

`gslice::size`

Finds the array values specifying the numbers of elements in a general slice of a `valarray`.

C++

```
valarray<size_t> size() const;
```

Return value

A `valarray` specifying the number of elements in each slice of a general slice of a `valarray`.

Remarks

The member function returns the stored lengths of slices.

Example

C++

```
// gslice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   int i;
   size_t sizeVA;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   sizeVA = va.size ( );
   cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
   const valarray <size_t> sizeGS = vaGSlice.size ( );

   cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   cout << "The size of vaResult is:"
        << "\n vaGSlice.size ( ) = ( ";
      for ( i = 0 ; i < 2 ; i++ )
         cout << sizeGS[ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The size of the valarray is: 20.

The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The size of vaResult is:
vaGSlice.size ( ) = ( 4 4 ).
```

`gslice::start`

Finds the starting index of a general slice of a `valarray`.

C++

```
size_t start() const;
```

Return value

The starting index of a general slice of a `valarray`.

Example

C++

```
// gslice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for (i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   valarray<size_t> Len ( 2 ), Stride ( 2 );
   Len [0] = 4;
   Len [1] = 4;
   Stride [0] = 7;
   Stride [1] = 4;

   gslice vaGSlice ( 0, Len, Stride );
   vaResult = va [ vaGSlice ];
   size_t vaGSstart = vaGSlice.start ( );

   cout << "The valarray for vaGSlice is vaResult:"
        << "\n va[vaGSlice] = ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaResult [ i ] << " ";
```

```
   cout << ")." << endl;

   cout << "The index of the first element of vaResult is: "
        << vaGSstart << "." << endl;
}
```

Output

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The index of the first element of vaResult is: 0.
```

`gslice::stride`

Finds the distance between elements in a general slice of a `valarray`.

C++

```
valarray<size_t> stride() const;
```

Return value

A `valarray` specifying the distances between elements in each slice of a general slice of a `valarray`.

Example

C++

```
// gslice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for (i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  i;
```

```
    cout << "The operand valarray va is:\n ( ";
       for (i = 0 ; i < 20 ; i++ )
          cout << va [ i ] << " ";
    cout << ")." << endl;

    valarray<size_t> Len ( 2 ), Stride ( 2 );
    Len [0] = 4;
    Len [1] = 4;
    Stride [0] = 7;
    Stride [1] = 4;

    gslice vaGSlice ( 0, Len, Stride );
    vaResult = va [ vaGSlice ];
    const valarray <size_t> strideGS = vaGSlice.stride ( );

    cout << "The valarray for vaGSlice is vaResult:"
         << "\n va[vaGSlice] = ( ";
       for ( i = 0 ; i < 8 ; i++ )
          cout << vaResult [ i ] << " ";
    cout << ")." << endl;

    cout << "The strides of vaResult are:"
         << "\n vaGSlice.stride ( ) = ( ";
       for ( i = 0 ; i < 2 ; i++ )
          cout << strideGS[ i ] << " ";
    cout << ")." << endl;

}
```

Output

```
The operand valarray va is:
( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ).
The valarray for vaGSlice is vaResult:
va[vaGSlice] = ( 0 4 8 12 7 11 15 19 ).
The strides of vaResult are:
vaGSlice.stride ( ) = ( 7 4 ).
```

See also

Thread safety in the C++ Standard Library

`gslice_array` class

Article • 05/12/2022

An internal, auxiliary class template that supports general slice objects by providing operations between subset arrays defined by the general slice of a `valarray`.

Syntax

C++

```
template <class Type>
class gslice_array : public gsplice {
public:
    typedef Type value_type;
    void operator=(const valarray<Type>& x) const;

    void operator=(const Type& x) const;

    void operator*=(const valarray<Type>& x) const;

    void operator/=(const valarray<Type>& x) const;

    void operator%=(const valarray<Type>& x) const;

    void operator+=(const valarray<Type>& x) const;

    void operator-=(const valarray<Type>& x) const;

    void operator^=(const valarray<Type>& x) const;

    void operator&=(const valarray<Type>& x) const;

    void operator|=(const valarray<Type>& x) const;

    void operator<<=(const valarray<Type>& x) const;

    void operator>>=(const valarray<Type>& x) const;

// The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object `va` of class valarray<Type>, along with an object `gs` of class gslice, which describes the sequence of elements to select from the `valarray<Type>` object.

You construct a `gslice_array<Type>` object only by writing an expression of the form va[gs]. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The class template is created indirectly by certain `valarray` operations and can't be used directly in the program. An internal auxiliary class template instead is used by the `slice` subscript operator:

`gslice_array<Type> valarray<Type>::operator[]( const gslice& )`.

You construct a `gslice_array<Type>` object only by writing an expression of the form `va[gsl]`, for a `slice` `gsl` of `valarray` `va`. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected. The sequence controlled by the `gslice_array` is defined by the three parameters of the `slice` constructor, the index of the first element in the first `slice`, the number of elements in each `slice`, and the distance between the elements in each `slice`.

In the following example:

C++

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);

// va[gslice(3, len, str)] selects elements with
//   indices 3, 5, 7, 10, 12, 14
```

The indices must be valid for the procedure to be valid.

Example

See the example for gslice::gslice for an example of how to declare and use a slice_array.

Requirements

Header: <valarray>

Namespace: `std`

See also

Thread safety in the C++ Standard Library

`indirect_array` class

Article • 05/12/2022

An internal, auxiliary class template that supports objects that are subsets of `valarray` objects by providing operations between subset arrays defined by specifying a subset of indices of a parent `valarray`.

Syntax

Remarks

The class describes an object that stores a reference to an object `va` of class valarray<Type>, along with an object `xa` of class `valarray<size_t>`, which describes the sequence of elements to select from the `valarray<Type>` object.

You construct an `indirect_array<Type>` object only by writing an expression of the form `va[xa]`. The member functions of class `indirect_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The sequence consists of xa.size elements, where element `I` becomes the index `xa[I]` within `va`.

Example:

C++

```
// indirect_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      va [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      va [ i ] =  -1;
```

```
   cout << "The initial operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   // Select 2nd, 4th & 6th elements
   // and assign a value of 10 to them
   valarray<size_t> indx ( 3 );
   indx [0] = 2;
   indx [1] = 4;
   indx [2] = 6;
   va[indx] = 10;

   cout << "The modified operand valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;
}
```

Output

C++

```
The initial operand valarray is:  (0 -1 2 -1 4 -1 6 -1 8 -1).
The modified operand valarray is:  (0 -1 10 -1 10 -1 10 -1 8 -1).
```

Requirements

Header: <valarray>

Namespace: `std`

See also

Thread safety in the C++ Standard Library

`mask_array` class

Article • 05/12/2022

An internal, auxiliary class template that supports objects that are subsets of parent `valarray` objects, specified with a Boolean expression, by providing operations between the subset arrays.

Remarks

The class describes an object that stores a reference to an object `va` of class valarray<Type>, along with an object `ba` of class valarray<bool>, which describes the sequence of elements to select from the `valarray<Type>` object.

You construct a `mask_array<Type>` object only by writing an expression of the form va[ba]. The member functions of class `mask_array` then behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected.

The sequence consists of at most `ba.size` elements. An element `J` is included only if `ba[J]` is `true`. There are as many elements in the sequence as there are `true` elements in `ba`. If `I` is the index of the lowest true element in `ba`, then `va[I]` is element zero in the selected sequence.

Example

C++

```
// mask_array.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      va [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 2 )
      va [ i ] =  -1;
```

```
   cout << "The initial operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   // Use masked subsets to assign a value of 10
   // to all elements grrater than 3 in value
   va [va > 3 ] = 10;
   cout << "The modified operand valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;
}
```

Output

Output

```
The initial operand valarray is:  (0 -1 2 -1 4 -1 6 -1 8 -1).
The modified operand valarray is:  (0 -1 2 -1 10 -1 10 -1 10 -1).
```

Requirements

Header: <valarray>

Namespace: `std`

See also

Thread Safety in the C++ Standard Library

`slice` class

Article • 05/12/2022

A utility class to `valarray` that's used to define one-dimensional subsets of a parent `valarray`. If a `valarray` is regarded as a two-dimensional matrix with all elements in an array, then the `slice` extracts a vector in one dimension out of the two-dimensional array.

Remarks

The class stores the parameters that characterize an object of type slice_array. The subset of a `valarray` is indirectly constructed when an object of class `slice` appears as an argument for an object of class valarray<Type>. The stored values that specify the subset selected from the parent `valarray` include:

- A starting index in the `valarray`.
- A total length, or number of elements in the `slice`.
- A stride, or distance between the indices of elements in the `valarray`.

If the set defined by a `slice` is the subset of a constant `valarray`, then the `slice` is a new `valarray`. If the set defined by a `slice` is the subset of a nonconstant `valarray`, then the `slice` has reference semantics to the original `valarray`. The evaluation mechanism for nonconstant `valarray` objects saves time and memory.

Operations on `valarray` objects are guaranteed only if the source and destination subsets defined by the `slice` objects are distinct and all indices are valid.

Constructors

Constructor	Description
slice	Defines a subset of a `valarray` that consists of a collection of elements that are an equal distance apart and that start at a specified element.

Member functions

Member function	Description

Member function	Description
size	Finds the number of elements in a `slice` of a `valarray`.
start	Finds the starting index of a `slice` of a `valarray`.
stride	Finds the distance between elements in a `slice` of a `valarray`.

Requirements

Header: <valarray>

Namespace: `std`

`slice::size`

Finds the number of elements in a `slice` of a `valarray`.

C++

```
size_t size() const;
```

Return value

The number of elements in a `slice` of a `valarray`.

Example

C++

```
// slice_size.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t sizeVA, sizeVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] =  i+1;
```

```
   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   sizeVA = va.size ( );
   cout << "The size of the valarray is: "
        << sizeVA << "." << endl << endl;

   slice vaSlice ( 3 , 6 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   sizeVAR = vaSlice.size ( );
   cout << "The size of slice vaSlice is: "
        << sizeVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The size of the valarray is: 20.

The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
( 4 7 10 13 16 19 ).
The size of slice vaSlice is: 6.
```

`slice::slice`

Defines a subset of a `valarray` that consists of a collection of elements that are an equal distance apart and that start at a specified element.

C++

```
slice();

slice(
    size_t _StartIndex,
    size_t _Len,
    size_t stride);
```

Parameters

_StartIndex

The valarray index of the first element in the subset.

_Len

The number of elements in the subset.

stride

The distance between elements in the subset.

Return value

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores *_StartIndex* for the starting index, *_Len* for the total length, and *stride* for the stride.

Remarks

The *stride* value may be negative.

Example

C++

```
// slice_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i+=1 )
      va [ i ] =  2 * (i + 1 );

   cout << "The operand valarray va is:\n( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   slice vaSlice ( 1 , 7 , 3 );
   vaResult = va [ vaSlice ];

   cout << "\nThe slice of valarray va is vaResult:"
        << "\nva[slice( 1, 7, 3)] = ( ";
```

```
      for ( i = 0 ; i < 7 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The operand valarray va is:
( 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 ).

The slice of valarray va is vaResult:
va[slice( 1, 7, 3)] = ( 4 10 16 22 28 34 40 ).
```

`slice::start`

Finds the starting index of a `slice` of a `valarray`.

C++

```
size_t start() const;
```

Return value

The starting index of a `slice` of a `valarray`.

Example

C++

```
// slice_start.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
   size_t startVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] = i+1;

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
```

```
         cout << va [ i ] << " ";
   cout << ")." << endl;

   slice vaSlice ( 3 , 6 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 3, 6, 3)] =\n ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   startVAR = vaSlice.start ( );
   cout << "The start index of slice vaSlice is: "
        << startVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ).
The slice of valarray va is vaResult = va[slice( 3, 6, 3)] =
( 4 7 10 13 16 19 ).
The start index of slice vaSlice is: 3.
```

`slice::stride`

Finds the distance between elements in a `slice` of a `valarray`.

C++

```
size_t stride() const;
```

Return value

The distance between elements in a `slice` of a `valarray`.

Example

C++

```
// slice_stride.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   int i;
   size_t strideVAR;

   valarray<int> va ( 20 ), vaResult;
   for ( i = 0 ; i < 20 ; i += 1 )
      va [ i ] =  3 * ( i + 1 );

   cout << "The operand valarray va is:\n ( ";
      for ( i = 0 ; i < 20 ; i++ )
         cout << va [ i ] << " ";
   cout << ")." << endl;

   slice vaSlice ( 4 , 5 , 3 );
   vaResult = va [ vaSlice ];

   cout << "The slice of valarray va is vaResult = "
        << "va[slice( 4, 5, 3)] =\n ( ";
      for ( i = 0 ; i < 5 ; i++ )
         cout << vaResult [ i ] << " ";
   cout << ")." << endl;

   strideVAR = vaSlice.stride ( );
   cout << "The stride of slice vaSlice is: "
        << strideVAR << "." << endl;
}
```

Output

```
The operand valarray va is:
( 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 ).
The slice of valarray va is vaResult = va[slice( 4, 5, 3)] =
( 15 24 33 42 51 ).
The stride of slice vaSlice is: 3.
```

See also

Thread safety in the C++ Standard Library

`slice_array` class

Article • 05/12/2022

An internal, auxiliary class template that supports `slice` objects by providing operations between subset arrays defined by the slice of a `valarray`.

Syntax

C++

```
template <class Type>
class slice_array : public slice {
public:
    typedef Type value_type;
    void operator=(const valarray<Type>& x) const;
    void operator=(const Type& x) const;
    void operator*=(const valarray<Type>& x) const;
    void operator/=(const valarray<Type>& x) const;
    void operator%=(const valarray<Type>& x) const;
    void operator+=(const valarray<Type>& x) const;
    void operator-=(const valarray<Type>& x) const;
    void operator^=(const valarray<Type>& x) const;
    void operator&=(const valarray<Type>& x) const;
    void operator|=(const valarray<Type>& x) const;
    void operator<<=(const valarray<Type>& x) const;
    void operator>>=(const valarray<Type>& x) const;
    // The rest is private or implementation defined
}
```

Remarks

The class describes an object that stores a reference to an object of class valarray<Type>, along with an object of class slice, which describes the sequence of elements to select from the `valarray<Type>` object.

The class template is created indirectly by certain `valarray` operations and can't be used directly in the program. An internal, auxiliary class template that is used by the `slice` subscript operator:

`slice_array<Type> valarray<Type>::operator[]( slice )`.

You construct a `slice_array<Type>` object only by writing an expression of the form va[sl], for a slice `sl` of valarray `va`. The member functions of class `slice_array` then

behave like the corresponding function signatures defined for `valarray<Type>`, except that only the sequence of selected elements is affected. The sequence controlled by the `slice_array` is defined by the three parameters of the `slice` constructor, the index of the first element in the `slice`, the number of elements, and the distance between the elements. A `slice_array` cut from `valarray` `va` declared by `va[slice(2, 5, 3)]` selects elements with indices 2, 5, 8, 11, and 14 from `va`. The indices must be valid for the procedure to be valid.

Example

See the example for slice::slice for an example of how to declare and use a `slice_array`.

Requirements

Header: <valarray>

Namespace: `std`

See also

Thread safety in the C++ Standard Library

`valarray` class

Article • 06/21/2022

The class template `valarray<Type>` describes an object that controls a sequence of elements of type `Type` that are stored as an array, designed for high-speed mathematical operations, and optimized for computational performance.

Remarks

The class is a representation of the mathematical concept of an ordered set of values. Its elements are numbered sequentially from zero. The class is described as a *near container* because it supports some, but not all, of the capabilities that first-class sequence containers, such as vector, support. It differs from class template `vector` in two important ways:

- It defines many arithmetic operations between corresponding elements of `valarray<Type>` objects of the same type and length, such as `varr_x* = cos(varr_y) + sin(varr_z)`.
- It defines various interesting ways to subscript a `valarray<Type>` object, by overloading operator[].

An object of class `Type`:

- Has a public default constructor, destructor, copy constructor, and assignment operator, with conventional behavior.
- Defines the arithmetic operators and math functions, as needed, that are defined for the floating-point types, with conventional behavior.

In particular, no subtle differences may exist between copy construction and default construction followed by assignment. None of the operations on objects of class `Type` may throw exceptions.

Members

Constructors

Name	Description

Name	Description
valarray	Constructs a `valarray` of a specific size or with elements of a specific value or as a copy of another `valarray` or subset of another `valarray`.

Typedefs

Name	Description
value_type	A type that represents the type of element stored in a `valarray`.

Functions

Name	Description
apply	Applies a specified function to each element of a `valarray`.
cshift	Cyclically shifts all the elements in a `valarray` by a specified number of positions.
free	Frees the memory used by the `valarray`.
max	Finds the largest element in a `valarray`.
min	Finds the smallest element in a `valarray`.
resize	Changes the number of elements in a `valarray` to a specified number, adding or removing elements as required.
shift	Shifts all the elements in a `valarray` by a specified number of positions.
size	Finds the number of elements in a `valarray`.
sum	Determines the sum of all the elements in a `valarray` of nonzero length.
swap	

Operators

Name	Description
operator!	A unary operator that obtains the logical NOT (`!`) values of each element in a `valarray`.
operator%=	Obtains the remainder of dividing the elements of an array element-wise either by a specified `valarray` or by a value of the element type.

Name	Description
operator&=	Obtains the bitwise AND (`&`) of elements in an array either with the corresponding elements in a specified `valarray` or with a value of the element type.
operator>>=	Right-shifts the bits for each element of a `valarray` operand a specified number of positions or by an element-wise amount specified by a second `valarray`.
operator<<=	Left-shifts the bits for each element of a `valarray` operand a specified number of positions or by an element-wise amount specified by a second `valarray`.
operator*=	Multiplies the elements of a specified `valarray` or a value of the element type, element-wise, to an operand `valarray`.
operator+	A unary operator that applies a plus to each element in a `valarray`. On normal arithmetic types, this operator has no effect.
operator+=	Adds the elements of a specified `valarray` or a value of the element type, element-wise, to an operand `valarray`.
operator-	A unary operator that applies an arithmetic negation to each element in a `valarray`.
operator-=	Subtracts the elements of a specified `valarray` or a value of the element type, element-wise, from an operand `valarray`.
operator/=	Divides an operand `valarray` element-wise by the elements of a specified `valarray` or a value of the element type.
operator=	Assigns elements to a `valarray` whose values are specified either directly or as part of some other `valarray` or by a `slice_array`, `gslice_array`, `mask_array`, or `indirect_array`.
operator[]	Returns a reference to an element or its value at specified index or a specified subset.
operator^=	Obtains the bitwise exclusive or (`^`) of a `valarray` with either a specified `valarray` or a value of the element type.
operator|=	Obtains the bitwise OR (`|`) of elements in an array either with the corresponding elements in a specified `valarray` or with a value of the element type.
operator~	A unary operator that obtains the bitwise complement (`~`) values of each element in a `valarray`.

`apply`

Applies a specified function to each element of a `valarray`.

C++

```
valarray<Type> apply(Type function_object(Type)) const;

valarray<Type> apply(Type function_object(const Type&)) const;
```

Parameters

`function_object(Type)`
The function object to be applied to each element of the operand `valarray`.

`function_object(const Type&)`
The function object for `const` elements to be applied to each element of the operand `valarray`.

Return value

A `valarray` whose elements have had `function_object` applied element-wise to the elements of the operand `valarray`.

Remarks

The member function returns an object of class valarray<Type>, of length size, each of whose elements `elem` is the result of `function_object((*this)[elem])`.

Example

C++

```
// valarray_apply.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

using namespace std;

int __cdecl MyApplyFunc( int n )
{
   return n*2;
}

int main( int argc, char* argv[] )
{
   valarray<int> vaR(10), vaApplied(10);
```

```
    int i;

    for ( i = 0; i < 10; i += 3 )
       vaR[i] = i;

    for ( i = 1; i < 10; i += 3 )
       vaR[i] = 0;

    for ( i = 2; i < 10; i += 3 )
       vaR[i] = -i;

    cout << "The initial Right valarray is: (";
    for   ( i=0; i < 10; ++i )
       cout << " " << vaR[i];
    cout << " )" << endl;

    vaApplied = vaR.apply( MyApplyFunc );

    cout << "The element-by-element result of "
        << "applying MyApplyFunc to vaR is the\nvalarray: ( ";
    for ( i = 0; i < 10; ++i )
       cout << " " << vaApplied[i];
    cout << " )" << endl;
}
```

Output

```
The initial Right valarray is: ( 0 0 -2 3 0 -5 6 0 -8 9 )
The element-by-element result of applying MyApplyFunc to vaR is the
valarray: (  0 0 -4 6 0 -10 12 0 -16 18 )
```

`cshift`

Cyclically shifts all the elements in a `valarray` by a specified number of positions.

C++

```
valarray<Type> cshift(int count) const;
```

Parameters

`count`

The number of places the elements are to be shifted forward.

Return value

A new `valarray` in which all the elements have been moved *`count`* positions cyclically toward the front of the `valarray`, or leftward compared to their positions in the operand `valarray`.

Remarks

A positive value of *`count`* shifts the elements cyclically left *`count`* places.

A negative value of *`count`* shifts the elements cyclically right *`count`* places.

Example

C++

```
// valarray_cshift.cpp
// compile with: /EHsc

#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;

    valarray<int> va1(10), va2(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;
    for (i = 0; i < 10; i+=1)
        va2[i] = 10 - i;

    cout << "The operand valarray va1 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    // A positive parameter shifts elements right
    va1 = va1.cshift(4);
    cout << "The cyclically shifted valarray va1 is:\nva1.cshift (4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va1[i];
    cout << ")" << endl;

    cout << "The operand valarray va2 is: (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;

    // A negative parameter shifts elements left
    va2 = va2.cshift(-4);
```

```
    cout << "The cyclically shifted valarray va2 is:\nva2.shift (-4) = (";
    for (i = 0; i < 10; i++)
        cout << " " << va2[i];
    cout << ")" << endl;
}
```

Output

```
The operand valarray va1 is: ( 0 1 2 3 4 5 6 7 8 9)
The cyclically shifted valarray va1 is:
va1.cshift (4) = ( 4 5 6 7 8 9 0 1 2 3)
The operand valarray va2 is: ( 10 9 8 7 6 5 4 3 2 1)
The cyclically shifted valarray va2 is:
va2.shift (-4) = ( 4 3 2 1 10 9 8 7 6 5)
```

`free`

Frees the memory used by the `valarray`.

C++

```
void free();
```

Remarks

This nonstandard function is equivalent to assigning an empty `valarray`. For example:

C++

```
valarray<T> v;
v = valarray<T>();

// equivalent to v.free()
```

`max`

Finds the largest element in a `valarray`.

C++

```
Type max() const;
```

Return value

The maximum value of the elements in the operand `valarray`.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class `Type`, for which operators must be provided for the element `Type`.

Example

C++

```
// valarray_max.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i, MaxValue;

   valarray<int> vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  2*i - 1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  10 - i;

   cout << "The operand valarray is: ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   MaxValue = vaR.max (  );
   cout << "The largest element in the valarray is: "
        << MaxValue  << "." << endl;
}
```

Output

```
The operand valarray is: ( 0 1 8 3 7 5 6 13 2 9 ).
The largest element in the valarray is: 13.
```

`min`

Finds the smallest element in a `valarray`.

```
C++
```

```
Type min() const;
```

Return value

The minimum value of the elements in the operand `valarray`.

Remarks

The member function compares values by applying **operator<** or **operator>** between pairs of elements of class `Type`, for which operators must be provided for the element `Type`.

Example

```
C++
```

```
// valarray_min.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i, MinValue;

   valarray<int> vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  2*i;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  5 - i;

   cout << "The operand valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   MinValue = vaR.min ( );
```

```
    cout << "The smallest element in the valarray is: "
         << MinValue  << "." << endl;
}
/* Output:
The operand valarray is: ( 0 2 3 -3 8 0 -6 14 -3 -9 ).
The smallest element in the valarray is: -9.
*/
```

`operator!`

A unary operator that obtains the logical NOT (`!`) values of each element in a `valarray`.

C++

```
valarray<bool> operator!() const;
```

Return value

The `valarray` of Boolean values that are the logical negation of the element values of the operand `valarray`.

Remarks

The logical operation NOT, denoted by `!`, logically negates the elements because it converts all zero values into **`true`** and all non-zero values into **`false`**. The returned `valarray` of Boolean values is of the same size as the operand `valarray`.

Example

C++

```
// valarray_op_lognot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 );
   valarray<bool> vaNOT ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
```

```
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   vaNOT = !vaL;
   cout << "The element-by-element result of "
        << "the logical NOT operator! is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaNOT [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The element-by-element result of the logical NOT operator! is the
valarray: ( 1 1 1 0 1 0 1 0 1 0 ).
```

operator%=

Obtains the remainder of dividing the elements of an array element-wise either by a specified `valarray` or by a value of the element type.

C++

```
valarray<Type>& operator%=(const valarray<Type>& right);

valarray<Type>& operator%=(const Type& right);
```

Parameters

right

The `valarray` or value of an element type identical to that of the operand `valarray` that is to divide, element-wise, the operand `valarray`.

Return value

A `valarray` whose elements are the remainder from the element-wise division of the operand `valarray` by *`right`*.

Example

C++

```
// valarray_class_op_rem.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 6 ), vaR ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  53;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -67;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  3*i+1;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial  right valarray is: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL %= vaR;
   cout << "The remainders from the element-by-element "
        << "division is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 53 -67 53 -67 53 -67 ).
The initial  right valarray is: ( 1 4 7 10 13 16 ).
The remainders from the element-by-element division is the
valarray: ( 0 -3 4 -7 1 -3 ).
```

`operator&=`

Obtains the bitwise AND (`&`) of elements in an array either with the corresponding elements in a specified `valarray` or with a value of the element type.

C++

```
valarray<Type>& operator&=(const valarray<Type>& right);

valarray<Type>& operator&=(const Type& right);
```

Parameters

`right`
The `valarray` or value of an element type identical to that of the operand `valarray` that is to be combined, element-wise, by the bitwise AND (`&`) with the operand `valarray`.

Return value

A `valarray` whose elements are the bitwise AND (`&`) of the operand `valarray` by *`right`*

Remarks

A bitwise operation can only be used to manipulate bits in integral data types, such as **`char`** and **`int`**. It doesn't work on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`**, or other, more complex data types.

The bitwise AND (`&`) applies to the data type on the level of the individual bits. Given bits `b1` and `b2`, `b1 & b2` is 1 if both bits are 1; 0 if at least one bit is 0.

Example

C++

```
// valarray_class_op_bitand.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
```

```
   valarray<int> vaL ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL &= vaR;
   cout << "The element-by-element result of "
        << "the bitwise AND operator&= is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 0 2 0 4 0 6 0 8 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The element-by-element result of the bitwise AND operator&= is the
valarray: ( 0 0 0 2 0 4 0 6 0 8 ).
```

`operator>>=`

Right-shifts the bits for each element of a `valarray` operand a specified number of positions or by an element-wise amount specified by a second `valarray`.

C++

```
valarray<Type>& operator>>=(const valarray<Type>& right);

valarray<Type>& operator>>=(const Type& right);
```

Parameters

right

The value indicating the amount of right shift or a `valarray` whose elements indicate the element-wise amount of right shift.

Return value

A `valarray` whose elements have been shifted right the amount specified in *right*.

Remarks

Signed numbers have their signs preserved.

Example

C++

```
// valarray_class_op_rs.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  64;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -64;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial operand valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The  right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL >>= vaR;
   cout << "The element-by-element result of "
        << "the right shift is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
```

```
        cout << vaL [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial operand valarray is: ( 64 -64 64 -64 64 -64 64 -64 ).
The  right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the right shift is the
valarray: ( 64 -32 16 -8 4 -2 1 -1 ).
```

`operator<<=`

Left-shifts the bits for each element of a `valarray` operand a specified number of positions or by an element-wise amount specified by a second `valarray`.

C++

```
valarray<Type>& operator<<=(const valarray<Type>& right);

valarray<Type>& operator<<=(const Type& right);
```

Parameters

`right`
The value indicating the amount of left shift or `valarray` whose elements indicate the element-wise amount of left shift.

Return value

A `valarray` whose elements have been shifted left the amount specified in *`right`*.

Remarks

Signed numbers have their signs preserved.

Example

C++

```cpp
// valarray_class_op_ls.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial operand valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The  right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL <<= vaR;
   cout << "The element-by-element result of "
        << "the left shift"
        << endl << "on the operand array is the valarray:"
        << endl << "( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial operand valarray is: ( 1 -1 1 -1 1 -1 1 -1 ).
The  right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the left shift
on the operand array is the valarray:
( 1 -2 4 -8 16 -32 64 -128 ).
```

operator*=

Multiplies the elements of a specified `valarray` or a value of the element type, element-wise, to an operand `valarray`.

C++

```
valarray<Type>& operator*=(const valarray<Type>& right);

valarray<Type>& operator*=(const Type& right);
```

Parameters

`right`

The `valarray` or value of an element type identical to that of the operand `valarray` that is to multiply, element-wise, the operand `valarray`.

Return value

A `valarray` whose elements are the element-wise product of the operand `valarray` and *`right`*.

Example

C++

```
// valarray_op_emult.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
```

```
   cout << "The initial Right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL *= vaR;
   cout << "The element-by-element result of "
        << "the multiplication is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
/* Output:
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the multiplication is the
valarray: ( 0 -1 4 -3 8 -5 12 -7 ).
*/
```

`operator+`

A unary operator that applies a plus to each element in a `valarray`. For normal arithmetic values, this operation has no effect.

C++

```
valarray<Type> operator+() const;
```

Return value

A `valarray` whose elements are the unary plus result of the operand array.

Example

C++

```
// valarray_op_eplus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;
```

```
   valarray<int> vaL ( 10 );
   valarray<int> vaPLUS ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   vaPLUS = +vaL;
   cout << "The element-by-element result of "
        << "the operator+ is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaPLUS [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator+ is the
valarray: ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
```

`operator+=`

Adds the elements of a specified `valarray` or a value of the element type, element-wise, to an operand `valarray`.

C++

```
valarray<Type>& operator+=(const valarray<Type>& right);

valarray<Type>& operator+=(const Type& right);
```

Parameters

`right`

The `valarray` or value of an element type identical to that of the operand `valarray` that is to be added, element-wise, to the operand `valarray`.

Return value

A `valarray` whose elements are the element-wise sum of the operand `valarray` and *`right`*.

Example

C++

```
// valarray_op_eadd.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  2;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  -1;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial  right valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL += vaR;
   cout << "The element-by-element result of "
        << "the sum is the"
        << endl << "valarray: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 2 -1 2 -1 2 -1 2 -1 ).
The initial  right valarray is: ( 0 1 2 3 4 5 6 7 ).
```

```
The element-by-element result of the sum is the
valarray: ( 2 0 4 2 6 4 8 6 ).
```

`operator-`

A unary operator that applies an arithmetic negation to each element in a `valarray`.

C++

```
valarray<Type> operator-() const;
```

Return value

A `valarray` whose elements are arithmetically negated from the elements of the operand `valarray`.

Example

C++

```
// valarray_op_eminus.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 );
   valarray<int> vaMINUS ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i-1;

   cout << "The initial valarray is:  ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   vaMINUS = -vaL;
   cout << "The element-by-element result of "
        << "the operator- is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 10 ; i++ )
```

```
        cout << vaMINUS [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial valarray is:  ( 0 0 -2 2 -4 4 -6 6 -8 8 ).
The element-by-element result of the operator- is the
valarray: ( 0 0 2 -2 4 -4 6 -6 8 -8 ).
```

`operator-=`

Subtracts the elements of a specified `valarray` or a value of the element type, element-wise, from an operand `valarray`.

C++

```
valarray<Type>& operator-=(const valarray<Type>& right);

valarray<Type>& operator-=(const Type& right);
```

Parameters

`right`

The `valarray` or value of an element type identical to that of the operand `valarray` that is to be subtracted, element-wise, from the operand `valarray`.

Return value

A `valarray` whose elements are the element-wise difference of the operand `valarray` and *`right`*.

Example

C++

```
// valarray_op_esub.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   int i;

   valarray<int> vaL ( 8 ), vaR ( 8 );
   for ( i = 0 ; i < 8 ; i += 2 )
      vaL [ i ] =  10;
   for ( i = 1 ; i < 8 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 8 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial  right valarray is: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL -= vaR;
   cout << "The element-by-element result of "
        << "the difference is the"
        << endl << "valarray: ( ";
      for ( i = 0 ; i < 8 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 10 0 10 0 10 0 10 0 ).
The initial  right valarray is: ( 0 1 2 3 4 5 6 7 ).
The element-by-element result of the difference is the
valarray: ( 10 -1 8 -3 6 -5 4 -7 ).
```

operator/=

Divides an operand `valarray` element-wise by the elements of a specified `valarray` or a value of the element type.

C++

```
valarray<Type>& operator/=(const valarray<Type>& right);

valarray<Type>& operator/=(const Type& right);
```

Parameters

right

The valarray or value of an element type identical to that of the operand valarray that is to be divided, element-wise, into the operand valarray.

Return value

A valarray whose elements are the element-wise quotient of the operand valarray divided by *right*.

Example

C++

```
// valarray_op_ediv.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<double> vaL ( 6 ), vaR ( 6 );
   for ( i = 0 ; i < 6 ; i += 2 )
      vaL [ i ] =  100;
   for ( i = 1 ; i < 6 ; i += 2 )
      vaL [ i ] =  -100;
   for ( i = 0 ; i < 6 ; i++ )
      vaR [ i ] =  2*i;

   cout << "The initial valarray is: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL /= vaR;
   cout << "The element-by-element result of "
        << "the quotient is the"
        << endl << "valarray: ( ";
      for (i = 0 ; i < 6 ; i++ )
         cout << vaL [ i ] << " ";
```

```
    cout << ")." << endl;
}
```

Output

```
The initial valarray is: ( 100 -100 100 -100 100 -100 ).
The initial Right valarray is: ( 0 2 4 6 8 10 ).
The element-by-element result of the quotient is the
valarray: ( inf -50 25 -16.6667 12.5 -10 ).
```

`operator=`

Assigns elements to a `valarray` whose values are specified either directly or as part of some other `valarray` or by a `slice_array`, `gslice_array`, `mask_array`, or `indirect_array`.

C++

```
valarray<Type>& operator=(const valarray<Type>& right);

valarray<Type>& operator=(valarray<Type>&& right);

valarray<Type>& operator=(const Type& val);

valarray<Type>& operator=(const slice_array<Type>& slicearray);

valarray<Type>& operator=(const gslice_array<Type>& gslicearray);

valarray<Type>& operator=(const mask_array<Type>& maskarray);

valarray<Type>& operator=(const indirect_array<Type>& indarray);
```

Parameters

`right`

The `valarray` to be copied into the operand `valarray`.

`val`

The value to be assigned to the elements of the operand `valarray`.

`slicearray`

The `slice_array` to be copied into the operand `valarray`.

gslicearray

The `gslice_array` to be copied into the operand `valarray`.

maskarray

The `mask_array` to be copied into the operand `valarray`.

indarray

The `indirect_array` to be copied into the operand `valarray`.

Return value

The first member operator replaces the controlled sequence with a copy of the sequence controlled by *right*.

The second member operator is the same as the first, but with an Rvalue reference declarator: &&.

The third member operator replaces each element of the controlled sequence with a copy of *val*.

The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by operator[].

If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

Remarks

If the length of the controlled sequence changes, normally the result is undefined. In this implementation, however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

Example

C++

```
// valarray_op_assign.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
```

```
   int i;

   valarray<int> va ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 1 )
      va [ i ] = i;
   for ( i = 0 ; i < 10 ; i+=1 )
      vaR [ i ] = 10 -  i;

   cout << "The operand valarray va is:";
   for ( i = 0 ; i < 10 ; i++ )
      cout << " " << va [ i ];
   cout << endl;

   cout << "The operand valarray vaR is:";
      for ( i = 0 ; i < 10 ; i++ )
         cout << " " << vaR [ i ];
   cout << endl;

   // Assigning vaR to va with the first member function
   va = vaR;
   cout << "The reassigned valarray va is:";
   for ( i = 0 ; i < 10 ; i++ )
      cout << " " << va [ i ];
   cout << endl;

   // Assigning elements of value 10 to va
   // with the second member function
   va = 10;
   cout << "The reassigned valarray va is:";
      for ( i = 0 ; i < 10 ; i++ )
         cout << " " << va [ i ];
   cout << endl;
}
```

Output

```
The operand valarray va is: 0 1 2 3 4 5 6 7 8 9
The operand valarray vaR is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 9 8 7 6 5 4 3 2 1
The reassigned valarray va is: 10 10 10 10 10 10 10 10 10 10
```

operator[]

Returns a reference to an element or its value at specified index or a specified subset.

C++

```
Type& operator[](size_t index);

slice_array<Type> operator[](slice slicearray);
```

```
gslice_array<Type> operator[](const gslice& gslicearray);

mask_array<Type> operator[](const valarray<bool>& boolarray);

indirect_array<Type> operator[](const valarray<size_t>& indarray);

Type operator[](size_t index) const;

valarray<Type> operator[](slice slicearray) const;

valarray<Type> operator[](const gslice& gslicearray) const;

valarray<Type> operator[](const valarray<bool>& boolarray) const;

valarray<Type> operator[](const valarray<size_t>& indarray) const;
```

Parameters

index
The index of the element to be assigned a value.

slicearray
A `slice_array` of a `valarray` that specifies a subset to be selected or returned to a new `valarray`.

gslicearray
A `gslice_array` of a `valarray` that specifies a subset to be selected or returned to a new `valarray`.

boolarray
A `bool_array` of a `valarray` that specifies a subset to be selected or returned to a new `valarray`.

indarray
An `indirect_array` of a `valarray` that specifies a subset to be selected or returned to a new `valarray`.

Return value

A reference to an element or its value at specified index or a specified subset.

Remarks

The member operator is overloaded to provide several ways to select sequences of elements from among the ones controlled by `*this`. The first group of five member operators work together with various overloads of operator= (and other assigning operators) to allow selective replacement (*slicing*) of the controlled sequence. The selected elements must exist.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error occurs if you attempt to access an element outside the bounds of the `valarray`. For more information, see Checked iterators.

Example

See the examples for slice::slice and gslice::gslice for an example of how to declare and use `operator[]`.

`operator^=`

Obtains the bitwise exclusive or (`^`) of a `valarray` with either a specified `valarray` or a value of the element type.

```
C++

valarray<Type>& operator^=(const valarray<Type>& right);

valarray<Type>& operator^=(const Type& right);
```

Parameters

`right`
The `valarray` or value of an element type identical to that of the operand `valarray` that is to be combined by the bitwise exclusive or (XOR) operator (`^`) with the operand `valarray`.

Return value

A `valarray` whose elements are the bitwise exclusive or (XOR) of the operand `valarray` and *`right`*.

Remarks

The bitwise exclusive or, referred to as XOR and denoted by operator `^`, has the following semantics: Given bits `b1` and `b2`, `b1 ^ b2` is 1 if exactly one of the elements is 1, and 0 if both elements are 0 or 1.

Example

C++

```
// valarray_op_exor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<int> vaL ( 10 ), vaR ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL [ i ] =  1;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL [ i ] =  0;
    for ( i = 0 ; i < 10 ; i += 3 )
        vaR [ i ] =  i;
    for ( i = 1 ; i < 10 ; i += 3 )
        vaR [ i ] =  i-1;
    for ( i = 2 ; i < 10 ; i += 3 )
        vaR [ i ] =  i-1;

    cout << "The initial operand valarray is:  ( ";
        for (i = 0 ; i < 10 ; i++ )
            cout << vaL [ i ] << " ";
    cout << ")." << endl;

    cout << "The  right valarray is: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaR [ i ] << " ";
    cout << ")." << endl;

    vaL ^= vaR;
    cout << "The element-by-element result of "
        << "the bitwise XOR operator^= is the"
        << endl << "valarray: ( ";
        for (i = 0 ; i < 10 ; i++ )
            cout << vaL [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial operand valarray is:  ( 1 0 1 0 1 0 1 0 1 0 ).
The  right valarray is: ( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise XOR operator^= is the
valarray: ( 1 0 0 3 2 4 7 6 6 9 ).
```

`operator|=`

Obtains the bitwise OR (`|`) of elements in an array either with the corresponding elements in a specified `valarray` or with a value of the element type.

C++

```
valarray<Type>& operator|=(const valarray<Type>& right);

valarray<Type>& operator|=(const Type& right);
```

Parameters

`right`
The `valarray` or value of an element type identical to that of the operand `valarray` that is to be combined, element-wise, by the bitwise OR (`|`) with the operand `valarray`.

Return value

A `valarray` whose elements are the element-wise bitwise OR (`|`) of the operand `valarray` by *`right`*.

Remarks

A bitwise operation can only be used to manipulate bits in integral data types such as **`char`** and **`int`**. It doesn't work on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`**, or other, more complex data types.

The bitwise OR (`|`) applies to the data type on the level of the individual bits. Given bits `b1` and `b2`, `b1 | b2` is 1 if at least one of the bits is 1; 0 if both bits are 0.

Example

C++

```cpp
// valarray_class_op_bitor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  1;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  0;
   for ( i = 0 ; i < 10 ; i += 3 )
      vaR [ i ] =  i;
   for ( i = 1 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;
   for ( i = 2 ; i < 10 ; i += 3 )
      vaR [ i ] =  i-1;

   cout << "The initial operand valarray is:"
        << endl << "( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The  right valarray is:"
        << endl << "( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaL |= vaR;
   cout << "The element-by-element result of "
        << "the bitwise OR"
        << endl << "operator|= is the valarray:"
        << endl << "( ";
      for (i = 0 ; i < 10 ; i++ )
         cout << vaL [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The initial operand valarray is:
( 1 0 1 0 1 0 1 0 1 0 ).
The  right valarray is:
( 0 0 1 3 3 4 6 6 7 9 ).
The element-by-element result of the bitwise OR
```

```
operator|= is the valarray:
( 1 0 1 3 3 4 7 6 7 9 ).
```

`operator~`

A unary operator that obtains the bitwise complement of values of each element in a `valarray`.

C++

```
valarray<Type> operator~() const;
```

Return value

The `valarray` of values that are the unary bitwise complement of the element values of the operand `valarray`.

Remarks

A bitwise operation can only manipulate bits in integral types, such as **`char`** and **`int`**. It doesn't work on **`float`**, **`double`**, **`long double`**, **`void`**, **`bool`**, or other, more complex data types.

The unary bitwise complement operator `~` applies to the data type at the level of the individual bits. Given bit `b`, `~b` is 1 if `b` is 0 and 0 if `b` is 1.

Example

C++

```
// valarray_op_bitnot.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;

    valarray<unsigned short int> vaL1 ( 10 );
    valarray<unsigned short int> vaNOT1 ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
```

```
        vaL1 [ i ] =  i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL1 [ i ] =  5*i;

    cout << "The initial valarray <unsigned short int> is:  ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaL1 [ i ] << " ";
    cout << ")." << endl;

    vaNOT1 = ~vaL1;
    cout << "The element-by-element result of "
        << "the bitwise NOT operator~ is the"
        << endl << "valarray: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaNOT1 [ i ] << " ";
    cout << ")." << endl << endl;

    valarray<int> vaL2 ( 10 );
    valarray<int> vaNOT2 ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        vaL2 [ i ] =  i;
    for ( i = 1 ; i < 10 ; i += 2 )
        vaL2 [ i ] =  -2 * i;

    cout << "The initial valarray <int> is:  ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaL2 [ i ] << " ";
    cout << ")." << endl;

    vaNOT2 = ~vaL2;
    cout << "The element-by-element result of "
        << "the bitwise NOT operator~ is the"
        << endl << "valarray: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaNOT2 [ i ] << " ";
    cout << ")." << endl;

    // The negative numbers are represented using
    // the two's complement approach, so adding one
    // to the flipped bits returns the negative elements
    vaNOT2 = vaNOT2 + 1;
    cout << "The element-by-element result of "
        << "adding one"
        << endl << "is the negative of the "
        << "original elements the"
        << endl << "valarray: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << vaNOT2 [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial valarray <unsigned short int> is:  ( 0 5 2 15 4 25 6 35 8 45 ).
The element-by-element result of the bitwise NOT operator~ is the
valarray: ( 65535 65530 65533 65520 65531 65510 65529 65500 65527 65490 ).

The initial valarray <int> is:  ( 0 -2 2 -6 4 -10 6 -14 8 -18 ).
The element-by-element result of the bitwise NOT operator~ is the
valarray: ( -1 1 -3 5 -5 9 -7 13 -9 17 ).
The element-by-element result of adding one
is the negative of the original elements the
valarray: ( 0 2 -2 6 -4 10 -6 14 -8 18 ).
```

`resize`

Changes the number of elements in a `valarray` to a specified number.

C++

```
void resize(
    size_t new_size);

void resize(
    size_t new_size,
    const Type val);
```

Parameters

`new_size`

The number of elements in the resized `valarray`.

`val`

The value to be given to the elements of the resized `valarray`.

Remarks

The first member function initializes elements with their default constructor.

Any pointers or references to elements in the controlled sequence are invalidated.

Example

The following example demonstrates the use of the `valarray::resize` member function.

C++

```cpp
// valarray_resize.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, sizeNew;

    valarray<int> va1(10);
    for (i = 0; i < 10; i+=1)
        va1[i] = i;

    cout << "The valarray contains ( ";
        for (i = 0; i < 10; i++)
            cout << va1[i] << " ";
    cout << ")." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray is: "
         << size1  << "." <<endl << endl;

    va1.resize(15, 10);

    cout << "The valarray contains ( ";
        for (i = 0; i < 15; i++)
            cout << va1[i] << " ";
    cout << ")." << endl;
    sizeNew = va1.size();
    cout << "The number of elements in the resized valarray is: "
         << sizeNew  << "." <<endl << endl;
}
```

Output

```
The valarray contains ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray is: 10.

The valarray contains ( 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 ).
The number of elements in the resized valarray is: 15.
```

shift

Shifts all the elements in a `valarray` by a specified number of places.

C++

```cpp
valarray<Type> shift(int count) const;
```

Parameters

count

The number of places the elements are to be shifted forward.

Return value

A new `valarray` in which all the elements have been moved *count* positions toward the front of the `valarray`, or leftward when compared to their positions in the operand `valarray`.

Remarks

A positive value of *count* shifts the elements left *count* places, with zero fill.

A negative value of *count* shifts the elements right *count* places, with zero fill.

Example

C++

```
// valarray_shift.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> va1 ( 10 ), va2 ( 10 );
   for ( i = 0 ; i < 10 ; i += 1 )
      va1 [ i ] =  i;
   for ( i = 0 ; i < 10 ; i += 1 )
      va2 [ i ] = 10 -  i;

   cout << "The operand valarray va1(10) is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va1 [ i ] << " ";
   cout << ")." << endl;

   // A positive parameter shifts elements left
   va1 = va1.shift ( 4 );
   cout << "The shifted valarray va1 is: va1.shift (4) = ( ";
```

```
      for ( i = 0 ; i < 10 ; i++ )
         cout << va1 [ i ] << " ";
   cout << ")." << endl;

   cout << "The operand valarray va2(10) is: ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va2 [ i ] << " ";
   cout << ")." << endl;

   // A negative parameter shifts elements right
   va2 = va2.shift ( - 4 );
   cout << "The shifted valarray va2 is: va2.shift (-4) = ( ";
      for ( i = 0 ; i < 10 ; i++ )
         cout << va2 [ i ] << " ";
   cout << ")." << endl;
}
```

Output

```
The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The shifted valarray va1 is: va1.shift (4) = ( 4 5 6 7 8 9 0 0 0 0 ).
The operand valarray va2(10) is: ( 10 9 8 7 6 5 4 3 2 1 ).
The shifted valarray va2 is: va2.shift (-4) = ( 0 0 0 0 10 9 8 7 6 5 ).
```

`size`

Finds the number of elements in a `valarray`.

C++

```
size_t size() const;
```

Return value

The number of elements in the operand `valarray`.

Example

The following example demonstrates the use of the `valarray::size` member function.

C++

```
// valarray_size.cpp
// compile with: /EHsc
#include <valarray>
```

```
#include <iostream>

int main()
{
    using namespace std;
    int i;
    size_t size1, size2;

    valarray<int> va1(10), va2(12);
    for (i = 0; i < 10; i += 1)
        va1[i] =  i;
    for (i = 0; i < 10; i += 1)
        va2[i] =  i;

    cout << "The operand valarray va1(10) is: ( ";
        for (i = 0; i < 10; i++)
            cout << va1[i] << " ";
    cout << ")." << endl;

    size1 = va1.size();
    cout << "The number of elements in the valarray va1 is: va1.size = "
         << size1  << "." <<endl << endl;

    cout << "The operand valarray va2(12) is: ( ";
        for (i = 0; i < 10; i++)
            cout << va2[i] << " ";
    cout << ")." << endl;

    size2 = va2.size();
    cout << "The number of elements in the valarray va2 is: va2.size = "
         << size2  << "." << endl << endl;

    // Initializing two more elements to va2
    va2[10] = 10;
    va2[11] = 11;
    cout << "After initializing two more elements,\n"
         << "the operand valarray va2(12) is now: ( ";
        for (i = 0; i < 12; i++)
            cout << va2[i] << " ";
    cout << ")." << endl;
    cout << "The number of elements in the valarray va2 is still: "
         << size2  << "." << endl;
}
```

Output

```
The operand valarray va1(10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va1 is: va1.size = 10.

The operand valarray va2(12) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The number of elements in the valarray va2 is: va2.size = 12.

After initializing two more elements,
```

```
the operand valarray va2(12) is now: ( 0 1 2 3 4 5 6 7 8 9 10 11 ).
The number of elements in the valarray va2 is still: 12.
```

sum

Determines the sum of all the elements in a `valarray` of nonzero length.

C++

```
Type sum() const;
```

Return value

The sum of the elements of the operand `valarray`.

Remarks

If the length is greater than one, the member function adds values to the sum by applying `operator+=` between pairs of elements of class `Type`. It means the operator needs be provided for elements of type `Type`.

Example

C++

```
// valarray_sum.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    int sumva = 0;

    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i+=1 )
        va [ i ] =  i;

    cout << "The operand valarray va (10) is: ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << va [ i ] << " ";
    cout << ")." << endl;
```

```
    sumva = va.sum ( );
    cout << "The sum of elements in the valarray is: "
        << sumva  << "." <<endl;
}
```

Output

```
The operand valarray va (10) is: ( 0 1 2 3 4 5 6 7 8 9 ).
The sum of elements in the valarray is: 45.
```

`swap`

Exchanges the elements of two `valarray` objects.

C++

```
void swap(valarray& right);
```

Parameters

`right`
A `valarray` providing the elements to be swapped.

Remarks

The member function swaps the controlled sequences between `*this` and *`right`*. It does so in constant time, it throws no exceptions, and it invalidates no references, pointers, or iterators to elements in the two controlled sequences.

`valarray`

Constructs a `valarray` of a specific size or with elements of a specific value or as a copy of another `valarray` or subset of another `valarray`.

C++

```
valarray();

explicit valarray(
    size_t count);

valarray(
```

```
    const Type& val,
    size_t count);

valarray(
    const Type* ptr,
    size_t count);

valarray(
    const valarray<Type>& right);

valarray(
    const slice_array<Type>& sliceArray);

valarray(
    const gslice_array<Type>& gsliceArray);

valarray(
    const mask_array<Type>& maskArray);

valarray(
    const indirect_array<Type>& indArray);

valarray(
    valarray<Type>&& tight);

valarray(
    initializer_list<Type> iList);
```

Parameters

count

The number of elements to be in the `valarray`.

val

The value to be used in initializing the elements in the `valarray`.

ptr

Pointer to the values to be used to initialize the elements in the `valarray`.

right

An existing `valarray` to initialize the new `valarray`.

sliceArray

A `slice_array` whose element values are to be used in initializing the elements of the `valarray` being constructed.

gsliceArray

A `gslice_array` whose element values are to be used in initializing the elements of the

`valarray` being constructed.

`maskArray`
A `mask_array` whose element values are to be used in initializing the elements of the `valarray` being constructed.

`indArray`
A `indirect_array` whose element values are to be used in initializing the elements of the `valarray` being constructed.

`iList`
The `initializer_list` containing the elements to copy.

Remarks

The first (default) constructor initializes the object to an empty `valarray`. The next three constructors each initialize the object to a `valarray` of *`count`* elements as follows:

- For explicit `valarray(size_t count)`, each element is initialized with the default constructor.
- For `valarray(const Type& val, count)`, each element is initialized with *`val`*.
- For `valarray(const Type* ptr, count)`, the element at position `I` is initialized with `ptr[I]`.

Each remaining constructor initializes the object to a `valarray<Type>` object determined by the subset specified in the argument.

The last constructor is the same as the next to last, but with an Rvalue reference declarator: &&.

Example

C++

```
// valarray_ctor.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main()
{
    using namespace std;
```

```
    int i;

    // The second member function
    valarray<int> va(10);
    for (auto i : va){
        va[i] = 2 * (i + 1);
    }

    cout << "The operand valarray va is:\n(";
    for (auto i : va) {
        cout << " " << va[i];
    }
    cout << " )" << endl;

    slice Slice(2, 4, 3);

    // The fifth member function
    valarray<int> vaSlice = va[Slice];

    cout << "The new valarray initialized from the slice is vaSlice =\n"
        << "va[slice( 2, 4, 3)] = (";
    for (int i = 0; i < 3; i++) {
        cout << " " << vaSlice[i];
    }
    cout << " )" << endl;

    valarray<int> va2{{ 1, 2, 3, 4 }};
    for (auto& v : va2) {
        cout << v << " ";
    }
    cout << endl;
}
```

Output

```
The operand valarray va is:
( 0 2 2 2 2 2 2 2 2 2 )
The new valarray initialized from the slice is vaSlice =
va[slice( 2, 4, 3)] = ( 0 0 0 )
1 2 3 4
```

value_type

A type that represents the type of element stored in a `valarray`.

C++

```
typedef Type value_type;
```

Remarks

The type is a synonym for the template parameter `Type`.

Example

C++

```
// valarray_value_type.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
    using namespace std;
    int i;
    valarray<int> va ( 10 );
    for ( i = 0 ; i < 10 ; i += 2 )
        va [ i ] =  i;
    for ( i = 1 ; i < 10 ; i += 2 )
        va [ i ] =  -1;

    cout << "The initial operand valarray is:  ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << va [ i ] << " ";
    cout << ")." << endl;

    // value_type declaration and initialization:
    valarray<int>::value_type Right = 10;

    cout << "The decalared value_type Right is: "
            << Right << endl;
    va *= Right;
    cout << "The resulting valarray is:  ( ";
        for ( i = 0 ; i < 10 ; i++ )
            cout << va [ i ] << " ";
    cout << ")." << endl;
}
```

Output

```
The initial operand valarray is:  ( 0 -1 2 -1 4 -1 6 -1 8 -1 ).
The decalared value_type Right is: 10
The resulting valarray is:  ( 0 -10 20 -10 40 -10 60 -10 80 -10 ).
```

See also

`valarray<bool>` Class

Article • 12/06/2021

A specialized version of the class template **valarray<Type>** to elements of type `bool`.

Syntax

C++

```
class valarray<bool>
```

Example

C++

```
// valarray_bool.cpp
// compile with: /EHsc
#include <valarray>
#include <iostream>

int main( )
{
   using namespace std;
   int i;

   valarray<int> vaL ( 10 ), vaR ( 10 );
   valarray<bool> vaBool ( 10 );
   for ( i = 0 ; i < 10 ; i += 2 )
      vaL [ i ] =  -i;
   for ( i = 1 ; i < 10 ; i += 2 )
      vaL [ i ] =  i;
   for ( i = 0 ; i < 10 ; i++ )
      vaR [ i ] =  i;

   cout << "The initial Left valarray is: ( ";
   for ( i = 0 ; i < 10 ; i++ )
      cout << vaL [ i ] << " ";
   cout << ")." << endl;

   cout << "The initial Right valarray is: ( ";
   for ( i = 0 ; i < 10 ; i++ )
      cout << vaR [ i ] << " ";
   cout << ")." << endl;

   vaBool = ( vaL < vaR );
   cout << "The result of the less-than comparison "
   << "test is the\nvalarray<bool>: ( ";
```

```
    for ( i = 0 ; i < 10 ; i++ )
       cout << vaBool [ i ] << " ";
    cout << ")." << endl;
}
/* Output:
The initial Left valarray is: ( 0 1 -2 3 -4 5 -6 7 -8 9 ).
The initial Right valarray is: ( 0 1 2 3 4 5 6 7 8 9 ).
The result of the less-than comparison test is the
valarray<bool>: ( 0 0 1 0 1 0 1 0 1 0 ).
*/
```

Requirements

Header: <valarray>

Namespace: std

See also

valarray Class

Thread Safety in the C++ Standard Library

<variant>

Article • 10/17/2022

A variant object holds and manages a value. If the variant holds a value, that value's type has to be one of the template argument types given to variant. These template arguments are called alternatives.

Requirements

Header: <variant>

Namespace: std

Members

Operators

Name	Description
operator==	Tests if the variant object on the left side of the operator is equal to the variant object on the right side.
operator!=	Tests if the variant object on the left side of the operator is not equal to the variant object on the right side.
operator<	Tests if the variant object on the left side of the operator is less than the variant object on the right side.
operator<=	Tests if the variant object on the left side of the operator is less than or equal to the variant object on the right side.
operator>	Tests if the variant object on the left side of the operator is greater than the variant object on the right side.
operator>=	Tests if the variant object on the left side of the operator is greater than or equal to the variant object on the right side.

Functions

Name	Description
get	Gets the variant of an object.

Name	Description
get_if	Gets the variant of an object if it exists.
holds_alternative	Return `true` if a variant exists.
swap	Swaps a **variant**.
visit	Moves to the next **variant**.

Classes

Name	Description
bad_variant_access	Objects thrown to report invalid accesses to the value of a variant object.
variant	An object to either hold a value of one of its alternative types, or no value.

Structs

Name	Description
hash	
monostate	An alternative type for a variant to make the variant type default constructible.
uses_allocator	
variant_alternative	Assists the variant objects.
variant_size	Assists the variant objects.

Objects

Name	Description
variant_npos	

See also

Header Files Reference

`<variant>` functions

Article • 12/07/2021

get

Gets the variant of an object.

C++

```
template <size_t I, class... Types>
    constexpr variant_alternative_t<I, variant<Types...>>&
get(variant<Types...>&);
template <size_t I, class... Types>
    constexpr variant_alternative_t<I, variant<Types...>>&&
get(variant<Types...>&&);
template <size_t I, class... Types>
    constexpr const variant_alternative_t<I, variant<Types...>>& get(const
variant<Types...>&);
template <size_t I, class... Types>
    constexpr const variant_alternative_t<I, variant<Types...>>&& get(const
variant<Types...>&&);
template <class T, class... Types>
    constexpr T& get(variant<Types...>&);
template <class T, class... Types>
    constexpr T&& get(variant<Types...>&&);
template <class T, class... Types>
    constexpr const T& get(const variant<Types...>&);
template <class T, class... Types>
    constexpr const T&& get(const variant<Types...>&&);
```

get_if

Gets the variant of an object if it exists.

C++

```
template <size_t I, class... Types>
    constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>>
get_if(variant<Types...>*) noexcept;
template <size_t I, class... Types>
    constexpr add_pointer_t<const variant_alternative_t<I,
variant<Types...>>> get_if(const variant<Types...>*) noexcept;
template <class T, class... Types>
    constexpr add_pointer_t<T> get_if(variant<Types...>*) noexcept;
template <class T, class... Types>
    constexpr add_pointer_t<const T> get_if(const variant<Types...>*)
noexcept;
```

holds_alternative

Return `true` if a variant exists.

```
C++
```

```
template <class T, class... Types>
    constexpr bool holds_alternative(const variant<Types...>&) noexcept;
```

swap

```
C++
```

```
template <class... Types>
    void swap(variant<Types...>&, variant<Types...>&) noexcept(see below);
```

variant_npos

```
C++
```

```
namespace std {
    inline constexpr size_t variant_npos = -1;
}
```

visit

Moves to the next **variant**.

```
C++
```

```
template <class Visitor, class... Variants>
    constexpr see below
        visit(Visitor&&, Variants&&...);
```

`<variant>` operators

Article • 12/07/2021

operator==

Tests if the forward list object on the left side of the operator is equal to the forward list object on the right side.

```
C++
```

```
template <class... Types>
    constexpr bool operator==(const variant<Types...>&, const
variant<Types...>&);
```

operator!=

Tests if the forward list object on the left side of the operator is not equal to the forward list object on the right side.

```
C++
```

```
template <class... Types>
    constexpr bool operator!=(const variant<Types...>&, const
variant<Types...>&);
```

`operator<`

Tests if the forward list object on the left side of the operator is less than the forward list object on the right side.

```
C++
```

```
template <class... Types>
    constexpr bool operator<(const variant<Types...>&, const
variant<Types...>&);
```

`operator<=`

Tests if the forward list object on the left side of the operator is less than or equal to the forward list object on the right side.

```
C++
```

```
template <class... Types>
    constexpr bool operator<=(const variant<Types...>&, const
variant<Types...>&);
```

operator>

Tests if the forward list object on the left side of the operator is greater than the forward list object on the right side.

```
C++
```

```
template <class... Types> constexpr
    bool operator>(const variant<Types...>&, const variant<Types...>&);
```

operator>=

Tests if the forward list object on the left side of the operator is greater than or equal to the forward list object on the right side.

```
C++
```

```
template <class... Types> constexpr
    bool operator>=(const variant<Types...>&, const variant<Types...>&);
```

variant Class

Article • 08/03/2021

Any instance of variant at any given time either holds a value of one of its alternative types, or it holds no value.

Syntax

```
C++
```

```
template <class... Types>
    class variant
```

Members

Constructors

Name	Description
variant	Constructs an object of type `variant`.

Functions

Name	Description
emplace	Creates a new contained value.
index	Returns the index of a contained value.
swap	
valueless_by_exception	Returns `false` if the variant holds a value.

Operators

Name	Description
operator=	Replaces the variant with a copy of another variant.

emplace

Creates a new contained value.

```
C++
```

```
template <class T, class... Args>
    T& emplace(Args&&...);
template <class T, class U, class... Args>
    T& emplace(initializer_list<U>, Args&&...);
template <size_t I, class... Args>
    variant_alternative_t<I, variant<Types...>>& emplace(Args&&...);
template <size_t I, class U, class... Args>
    variant_alternative_t<I, variant<Types...>>&
emplace(initializer_list<U>, Args&&...);
```

index

Returns the index of a contained value.

```
C++
```

```
constexpr size_t index() const noexcept;
```

variant

Constructs an object of type `variant`. Also includes a destructor.

```
C++
```

```
constexpr variant() noexcept(see below);
variant(const variant&);
variant(variant&&) noexcept(see below);
template <class T>
    constexpr variant(T&&) noexcept(see below);
template <class T, class... Args>
    constexpr explicit variant(in_place_type_t<T>, Args&&...);
template <class T, class U, class... Args>
    constexpr explicit variant(in_place_type_t<T>, initializer_list<U>,
Args&&...);
template <size_t I, class... Args>
    constexpr explicit variant(in_place_index_t<I>, Args&&...);
template <size_t I, class U, class... Args>
    constexpr explicit variant(in_place_index_t<I>, initializer_list<U>,
Args&&...);

template <class Alloc>
```

```
    variant(allocator_arg_t, const Al&);
template <class Alloc>
    variant(allocator_arg_t, const Al&, const variant&);
template <class Alloc>
    variant(allocator_arg_t, const Al&, variant&&);
template <class Alloc, class T>
    variant(allocator_arg_t, const Al&, T&&);
template <class Alloc, class T, class... Args>
    variant(allocator_arg_t, const Al&, in_place_type_t<T>, Args&&...);
template <class Alloc, class T, class U, class... Args>
    variant(allocator_arg_t, const Al&, in_place_type_t<T>,
initializer_list<U>, Args&&...);
template <class Alloc, size_t I, class... Args>
    variant(allocator_arg_t, const Al&, in_place_index_t<I>, Args&&...);
template <class Alloc, size_t I, class U, class... Args>
    variant(allocator_arg_t, const Al&, in_place_index_t<I>,
initializer_list<U>, Args&&...);

~variant();
```

Parameters

Al

The allocator class to use with this object.

operator=

Replaces the variant with a copy of another variant.

C++

```
variant& operator=(const variant&);
variant& operator=(variant&&) noexcept(see below);
template <class T>
    variant& operator=(T&&) noexcept(see below);
```

swap

C++

```
void swap(variant&) noexcept(see below);
```

valueless_by_exception

Returns `false` if the variant holds a value.

C++

```
constexpr bool valueless_by_exception() const noexcept;
```

<vector>

Article • 07/11/2022

Defines the container class template vector and several supporting templates.

The `vector` is a container that organizes elements of a given type in a linear sequence. It enables fast random access to any element, and dynamic additions and removals to and from the sequence. The `vector` is the preferred container for a sequence when random-access performance is at a premium.

> **Note**
>
> The `<vector>` library also uses the `#include <initializer_list>` statement.

For more information about the class `vector`, see vector Class. For information about the specialization `vector<bool>`, see vector<bool> class.

Syntax

C++

```
namespace std {
template <class Type, class Allocator>
class vector;
template <class Allocator>
class vector<bool>;

template <class Allocator>
struct hash<vector<bool, Allocator>>;

// TEMPLATE FUNCTIONS
template <class Type, class Allocator>
bool operator== (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator!= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator<(
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);
```

```
template <class Type, class Allocator>
bool operator> (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator<= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
bool operator>= (
    const vector<Type, Allocator>& left,
    const vector<Type, Allocator>& right);

template <class Type, class Allocator>
void swap (
    vector<Type, Allocator>& left,
    vector<Type, Allocator>& right);

}  // namespace std
```

Parameters

Type

The template parameter for the type of data stored in the vector.

Allocator

The template parameter for the stored allocator object responsible for memory allocation and deallocation.

left

The first (left) vector in a compare operation

right

The second (right) vector in a compare operation.

Members

Operators

Name	Description
operator! =	Tests if the `vector` object on the left side of the operator isn't equal to the `vector` object on the right side.

Name	Description
operator<	Tests if the `vector` object on the left side of the operator is less than the `vector` object on the right side.
operator<=	Tests if the `vector` object on the left side of the operator is less than or equal to the `vector` object on the right side.
operator==	Tests if the `vector` object on the left side of the operator is equal to the `vector` object on the right side.
operator>	Tests if the `vector` object on the left side of the operator is greater than the `vector` object on the right side.
operator>=	Tests if the `vector` object on the left side of the operator is greater than or equal to the `vector` object on the right side.

Classes

Name	Description
vector class	A class template of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element.

Specializations

Name	Description
hash	Returns a hash of the vector.
vector<bool> class	A full specialization of the class template vector for elements of type `bool` with an allocator for the underlying type used by the specialization.

Requirements

Header: `<vector>`

Namespace: `std`

See also

Header Files Reference

Thread Safety in the C++ Standard Library

`vector` class

Article • 08/25/2023

The C++ Standard Library vector class is a class template for sequence containers. A vector stores elements of a given type in a linear arrangement, and allows fast random access to any element. A vector is the preferred container for a sequence when random-access performance is at a premium.

Syntax

```
C++

template <class Type, class Allocator = allocator<Type>>
class vector
```

Parameters

`Type`
The element data type to be stored in the vector

`Allocator`
The type that represents the stored allocator object that encapsulates details about the vector's allocation and deallocation of memory. This argument is optional and the default value is `allocator<Type>`.

Remarks

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The deque class container is faster at insertions and deletions at the beginning and end of a sequence. The list class container is faster at insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The vector<bool> class is a full specialization of the class template vector for elements of type `bool`. It has an allocator for the underlying type used by the specialization.

The vector<bool> reference class is a nested class whose objects can provide references to elements (single bits) within a `vector<bool>` object.

Members

Constructors

Name	Description
vector	Constructs a vector of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other vector.

Typedefs

Name	Description
`[allocator_type]` (#allocator_type)	A type that represents the `allocator` class for the vector object.
const_iterator	A type that provides a random-access iterator that can read a `const` element in a vector.
const_pointer	A type that provides a pointer to a `const` element in a vector.
const_reference	A type that provides a reference to a `const` element stored in a vector. It's used for reading and doing `const` operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any `const` element in the vector.
difference_type	A type that provides the difference between the addresses of two elements in a vector.
iterator	A type that provides a random-access iterator that can read or modify any element in a vector.
pointer	A type that provides a pointer to an element in a vector.
reference	A type that provides a reference to an element stored in a vector.
reverse_iterator	A type that provides a random-access iterator that can read or modify any element in a reversed vector.

Name	Description
size_type	A type that counts the number of elements in a vector.
value_type	A type that represents the data type stored in a vector.

Functions

Name	Description
assign	Erases a vector and copies the specified elements to the empty vector.
at	Returns a reference to the element at a specified location in the vector.
back	Returns a reference to the last element of the vector.
begin	Returns a random-access iterator to the first element in the vector.
capacity	Returns the number of elements that the vector could contain without allocating more storage.
cbegin	Returns a random-access const iterator to the first element in the vector.
cend	Returns a random-access const iterator that points just beyond the end of the vector.
crbegin	Returns a const iterator to the first element in a reversed vector.
crend	Returns a const iterator to the end of a reversed vector.
clear	Erases the elements of the vector.
data	Returns a pointer to the first element in the vector.
emplace	Inserts an element constructed in place into the vector at a specified position.
emplace_back	Adds an element constructed in place to the end of the vector.
empty	Tests if the vector container is empty.
end	Returns a random-access iterator that points to the end of the vector.
erase	Removes an element or a range of elements in a vector from specified positions.
front	Returns a reference to the first element in a vector.
get_allocator	Returns an object to the `allocator` class used by a vector.
insert	Inserts an element or many elements into the vector at a specified position.
max_size	Returns the maximum length of the vector.

Name	Description
pop_back	Deletes the element at the end of the vector.
push_back	Add an element to the end of the vector.
rbegin	Returns an iterator to the first element in a reversed vector.
rend	Returns an iterator to the end of a reversed vector.
reserve	Reserves a minimum length of storage for a vector object.
resize	Specifies a new size for a vector.
shrink_to_fit	Discards excess capacity.
size	Returns the number of elements in the vector.
swap	Exchanges the elements of two vectors.

Operators

Name	Description
operator[]	Returns a reference to the vector element at a specified position.
operator=	Replaces the elements of the vector with a copy of another vector.

`allocator_type`

A type that represents the allocator class for the vector object.

```
C++
```

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter `Allocator`.

Example

See the example for get_allocator for an example that uses `allocator_type`.

`assign`

Erases a vector and copies the specified elements to the empty vector.

```
C++
```

```
void assign(size_type count, const Type& value);
void assign(initializer_list<Type> init_list);

template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Parameters

`first`

Position of the first element in the range of elements to be copied.

`last`

Position of the first element beyond the range of elements to be copied.

`count`

The number of copies of an element being inserted into the vector.

`value`

The value of the element being inserted into the vector.

`init_list`

The initializer_list containing the elements to insert.

Remarks

First, `assign` erases any existing elements in a vector. Then, `assign` either inserts a specified range of elements from the original vector into a vector, or it inserts copies of a new specified value element into a vector.

Example

```
C++
```

```
/ vector_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
```

```
int main()
{
    using namespace std;
    vector<int> v1, v2, v3;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
    v1.push_back(50);

    cout << "v1 = ";
    for (auto& v : v1){
        cout << v << " ";
    }
    cout << endl;

    v2.assign(v1.begin(), v1.end());
    cout << "v2 = ";
    for (auto& v : v2){
        cout << v << " ";
    }
    cout << endl;

    v3.assign(7, 4);
    cout << "v3 = ";
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;

    v3.assign({ 5, 6, 7 });
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;
}
```

at

Returns a reference to the element at a specified location in the vector.

C++

```
reference at(size_type position);

const_reference at(size_type position) const;
```

Parameters

position

The subscript or position number of the element to reference in the vector.

Return value

A reference to the element subscripted in the argument. If *position* is greater than the size of the vector, `at` throws an exception.

Remarks

If the return value of `at` is assigned to a `const_reference`, the vector object can't be modified. If the return value of `at` is assigned to a `reference`, the vector object can be modified.

Example

C++

```
// vector_at.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const int &i = v1.at( 0 );
   int &j = v1.at( 1 );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

back

Returns a reference to the last element of the vector.

```
C++
```

```
reference back();

const_reference back() const;
```

Return value

The last element of the vector. If the vector is empty, the return value is undefined.

Remarks

If the return value of `back` is assigned to a `const_reference`, the vector object can't be modified. If the return value of `back` is assigned to a `reference`, the vector object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error occurs if you attempt to access an element in an empty vector. For more information, see Checked iterators.

Example

```
C++
```

```
// vector_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.back( );
   const int& ii = v1.front( );

   cout << "The last integer of v1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of v1 is "<< ii << endl;
}
```

`begin`

Returns a random-access iterator to the first element in the vector.

C++

```
const_iterator begin() const;

iterator begin();
```

Return value

A random-access iterator addressing the first element in the `vector` or to the location succeeding an empty `vector`. Always compare the value returned with vector::end to ensure it's valid.

Remarks

If the return value of `begin` is assigned to a vector::const_iterator, the `vector` object can't be modified. If the return value of `begin` is assigned to an vector::iterator, the `vector` object can be modified.

Example

C++

```
// vector_begin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::iterator c1_Iter;
    vector<int>::const_iterator c1_cIter;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_Iter = c1.begin();
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
```

```
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_Iter = c1.begin();
    *c1_Iter = 20;
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}
```

Output

```
The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2
```

`capacity`

Returns the number of elements that the vector could contain without allocating more storage.

C++

```
size_type capacity() const;
```

Return value

The current length of storage allocated for the vector.

Remarks

The member function resize will be more efficient if sufficient memory is allocated to accommodate it. Use the member function reserve to specify the amount of memory allocated.

Example

C++

```
// vector_capacity.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 1 );
   cout << "The length of storage allocated is "
        << v1.capacity( ) << "." << endl;

   v1.push_back( 2 );
   cout << "The length of storage allocated is now "
        << v1.capacity( ) << "." << endl;
}
```

Output

```
The length of storage allocated is 1.
The length of storage allocated is now 2.
```

cbegin

Returns a `const` iterator that addresses the first element in the range.

C++

```
const_iterator cbegin() const;
```

Return value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range can't be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

```
C++

auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

cend

Returns a `const` past-the-end iterator that points to the element following the last element of the vector.

```
C++

const_iterator cend() const;
```

Return value

A `const` past-the-end iterator for the vector. It points to the element following the last element of the vector. That element is a placeholder and shouldn't be dereferenced. Only use it for comparisons. If the vector is empty, then `vector::cend() == vector::cbegin()`.

Remarks

`cend` is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

```
C++
```

```
auto i1 = Container.end();
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by `cend` shouldn't be dereferenced. Only use it for comparisons.

`clear`

Erases the elements of the vector.

C++

```
void clear();
```

Example

C++

```
// vector_clear.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "The size of v1 is " << v1.size( ) << endl;
   v1.clear( );
   cout << "The size of v1 after clearing is " << v1.size( ) << endl;
}
```

Output

```
The size of v1 is 3
The size of v1 after clearing is 0
```

`const_iterator`

A type that provides a random-access iterator that can read a **`const`** element in a vector.

```
C++
```

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` can't be used to modify the value of an element.

Example

See the example for back for an example that uses `const_iterator`.

`const_pointer`

A type that provides a pointer to a **`const`** element in a vector.

```
C++
```

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` can't be used to modify the value of an element.

An iterator is more commonly used to access a vector element.

`const_reference`

A type that provides a reference to a **`const`** element stored in a vector. It's used for reading and doing **`const`** operations.

```
C++
```

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` can't be used to modify the value of an element.

Example

C++

```
// vector_const_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const vector <int> v2 = v1;
   const int &i = v2.front( );
   const int &j = v2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error as v2 is const
   // v2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

`const_reverse_iterator`

A type that provides a random-access iterator that can read any **`const`** element in the vector.

C++

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` can't modify the value of an element and is used to iterate through the vector in reverse.

Example

See rbegin for an example of how to declare and use an iterator.

`crbegin`

Returns a const iterator to the first element in a reversed vector.

C++

```
const_reverse_iterator crbegin() const;
```

Return value

A const reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed `vector`.

Remarks

With the return value of `crbegin`, the `vector` object can't be modified.

Example

C++

```
// vector_crbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;
   vector <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
```

```
    v1.push_back( 2 );

    v1_Iter = v1.begin( );
    cout << "The first element of vector is "
         << *v1_Iter << "." << endl;

    v1_rIter = v1.crbegin( );
    cout << "The first element of the reversed vector is "
         << *v1_rIter << "." << endl;
}
```

Output

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

crend

Returns a `const` past-the-end reverse iterator that points to the element following the last element of the reversed vector.

C++

```
const_reverse_iterator crend() const;
```

Return value

A `const` reverse past-the-end iterator for the reversed vector. It points to the element following the last element of the reversed vector, which is the same as the element before the first element of the non-reversed vector. That element is a placeholder and shouldn't be dereferenced. Only use it for comparisons.

Remarks

`crend` is used with a reversed `vector` just as vector::cend is used with a `vector`.

With the return value of `crend` (suitably decremented), the `vector` object can't be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `vector`.

The value returned by `crend` shouldn't be dereferenced. Only use it for comparisons.

Example

C++

```
// vector_crend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

`data`

Returns a pointer to the first element in the vector.

C++

```
const_pointer data() const;

pointer data();
```

Return value

A pointer to the first element in the vector or to the location succeeding an empty `vector`.

Example

C++

```
// vector_data.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::pointer c1_ptr;
    vector<int>::const_pointer c1_cPtr;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_cPtr = c1.data();
    for (size_t n = c1.size(); 0 < n; --n, c1_cPtr++)
    {
        cout << " " << *c1_cPtr;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_ptr = c1.data();
    *c1_ptr = 20;
    for (size_t n = c1.size(); 0 < n; --n, c1_ptr++)
    {
        cout << " " << *c1_ptr;
    }
    cout << endl;
}
```

Output

```
The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2
```

`difference_type`

A type that provides the difference between two iterators that refer to elements within the same vector.

C++

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

A `difference_type` can also be described as the number of elements between two pointers, because a pointer to an element contains its address.

An iterator is more commonly used to access a vector element.

Example

C++

```
// vector_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <vector>
#include <algorithm>

int main( )
{
   using namespace std;

   vector <int> c1;
   vector <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );

   vector <int>::difference_type   df_typ1, df_typ2, df_typ3;

   df_typ1 = count( c1_Iter, c2_Iter, 10 );
   df_typ2 = count( c1_Iter, c2_Iter, 20 );
   df_typ3 = count( c1_Iter, c2_Iter, 30 );
   cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
   cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
   cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

`emplace`

Inserts an element constructed in place into the vector at a specified position.

```
C++
```

```
template <class... Types>
iterator emplace(
    const_iterator position,
    Types&&... args);
```

Parameters

`position`

The position in the vector where the first element is inserted.

`args`

Constructor arguments. The function infers which constructor overload to invoke based on the arguments provided.

Return value

The function returns an iterator that points to the position where the new element was inserted into the `vector`.

Remarks

Any insertion operation can be expensive, see vector class for a discussion of `vector` performance.

Example

```
C++
```

```
// vector_emplace.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;
```

```
   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

// initialize a vector of vectors by moving v1
   vector < vector <int> > vv1;

   vv1.emplace( vv1.begin(), move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      {
      cout << "vv1[0] =";
      for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
         cout << " " << *Iter;
      cout << endl;
      }
}
```

Output

```
v1 = 10 20 30
vv1[0] = 10 20 30
```

`emplace_back`

Adds an element constructed in place to the end of the vector.

C++

```
template <class... Types>
void emplace_back(Types&&... args);
```

Parameters

`args`

Constructor arguments. The function infers which constructor overload to invoke based on the arguments provided.

Example

C++

```
#include <vector>
struct obj
{
   obj(int, double) {}
};

int main()
{
   std::vector<obj> v;
   v.emplace_back(1, 3.14); // obj in created in place in the vector
}
```

`empty`

Tests if the vector is empty.

C++

```
bool empty() const;
```

Return value

`true` if the vector is empty; `false` if the vector isn't empty.

Example

C++

```
// vector_empty.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );

   if ( v1.empty( ) )
      cout << "The vector is empty." << endl;
   else
```

```
        cout << "The vector is not empty." << endl;
}
```

Output

```
The vector is not empty.
```

end

Returns a past-the-end iterator that points to the element following the last element of the vector.

C++

```
iterator end();

const_iterator end() const;
```

Return value

A past-the-end iterator for the vector. It points to the element following the last element of the vector. That element is a placeholder and shouldn't be dereferenced. Only use it for comparisons. If the vector is empty, then `vector::end() == vector::begin()`.

Remarks

If the return value of `end` is assigned to a variable of type `const_iterator`, the vector object can't be modified. If the return value of `end` is assigned to a variable of type `iterator`, the vector object can be modified.

Example

C++

```
// vector_end.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
int main( )
{
   using namespace std;
   vector <int> v1;
```

```
   vector <int>::iterator v1_Iter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
      cout << *v1_Iter << endl;
}
```

Output

```
1
2
```

erase

Removes an element or a range of elements in a vector from specified positions.

C++

```
iterator erase(
    const_iterator position);

iterator erase(
    const_iterator first,
    const_iterator last);
```

Parameters

position

Position of the element to be removed from the vector.

first

Position of the first element removed from the vector.

last

Position just beyond the last element removed from the vector.

Return value

An iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists.

Example

C++

```
// vector_erase.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );
   v1.push_back( 40 );
   v1.push_back( 50 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) + 1, v1.begin( ) + 3 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

Output

```
v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50
```

`front`

Returns a reference to the first element in a vector.

C++

```
reference front();

const_reference front() const;
```

Return value

A reference to the first element in the vector object. If the vector is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the vector object can't be modified. If the return value of `front` is assigned to a **`reference`**, the vector object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error occurs if you attempt to access an element in an empty vector. For more information, see Checked iterators.

Example

C++

```
// vector_front.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.front( );
   const int& ii = v1.front( );

   cout << "The first integer of v1 is "<< i << endl;
   // by incrementing i, we move the front reference to the second element
   i++;
```

```
   cout << "Now, the first integer of v1 is "<< i << endl;
}
```

get_allocator

Returns a copy of the allocator object used to construct the vector.

C++

```
Allocator get_allocator() const;
```

Return value

The allocator used by the vector.

Remarks

Allocators for the vector class specify how the class manages storage. The default allocators supplied with C++ Standard Library container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ feature.

Example

C++

```
// vector_get_allocator.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   vector<int> v1;
   vector<int, allocator<int> > v2 = vector<int, allocator<int> >
(allocator<int>( )) ;

   // v3 will use the same allocator class as v1
   vector <int> v3( v1.get_allocator( ) );

   vector<int>::allocator_type xvec = v3.get_allocator( );
```

```
    // You can now call functions on the allocator class used by vec
}
```

`insert`

Inserts an element, or many elements, or a range of elements into the vector at a specified position.

C++

```
iterator insert(
    const_iterator position,
    const Type& value);

iterator insert(
    const_iterator position,
    Type&& value);

void insert(
    const_iterator position,
    size_type count,
    const Type& value);

template <class InputIterator>
void insert(
    const_iterator position,
    InputIterator first,
    InputIterator last);
```

Parameters

`position`

The position in the vector where the first element is inserted.

`value`

The value of the element being inserted into the vector.

`count`

The number of elements being inserted into the vector.

`first`

The position of the first element in the range of elements to be copied.

`last`

The position of the first element beyond the range of elements to be copied.

Return value

The first two `insert` functions return an iterator that points to the position where the new element was inserted into the vector.

Remarks

As a precondition, *`first`* and *`last`* must not be iterators into the vector, or the behavior is undefined. Any insertion operation can be expensive, see vector class for a discussion of `vector` performance.

Example

C++

```
// vector_insert.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.insert( v1.begin( ) + 1, 40 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
   v1.insert( v1.begin( ) + 2, 4, 50 );

   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   const auto v2 = v1;
   v1.insert( v1.begin( )+1, v2.begin( )+2, v2.begin( )+4 );
```

```
   cout << "v1 =";
   for (Iter = v1.begin( ); Iter != v1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

// initialize a vector of vectors by moving v1
   vector < vector <int> > vv1;

   vv1.insert( vv1.begin(), move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      {
      cout << "vv1[0] =";
      for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
         cout << " " << *Iter;
      cout << endl;
      }
}
```

Output

```
v1 = 10 20 30
v1 = 10 40 20 30
v1 = 10 40 50 50 50 50 20 30
v1 = 10 50 50 40 50 50 50 50 20 30
vv1[0] = 10 50 50 40 50 50 50 50 20 30
```

`iterator`

A type that provides a random-access iterator that can read or modify any element in a vector.

C++

```
typedef implementation-defined iterator;
```

Remarks

A type `iterator` can be used to modify the value of an element.

Example

See the example for begin.

`max_size`

Returns the maximum length of the vector.

```
C++
```

```
size_type max_size() const;
```

Return value

The maximum possible length of the vector.

Example

```
C++
```

```
// vector_max_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   i = v1.max_size( );
   cout << "The maximum possible length of the vector is " << i << "." <<
endl;
}
```

operator[]

Returns a reference to the vector element at a specified position.

```
C++
```

```
reference operator[](size_type position);

const_reference operator[](size_type position) const;
```

Parameters

position

The position of the vector element.

Return value

If the position specified is greater than or equal to the size of the container, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the vector object can't be modified. If the return value of `operator[]` is assigned to a reference, the vector object can be modified.

When compiled by using _ITERATOR_DEBUG_LEVEL defined as 1 or 2, a runtime error occurs if you attempt to access an element outside the bounds of the vector. For more information, see Checked iterators.

Example

C++

```
// vector_op_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   int& i = v1[1];
   cout << "The second integer of v1 is " << i << endl;
}
```

`operator=`

Replaces the elements of the vector with a copy of another vector.

C++

```
vector& operator=(const vector& right);
```

```
vector& operator=(vector&& right);
```

Parameters

right

The vector being copied into the `vector`.

Remarks

After erasing any existing elements in a `vector`, `operator=` either copies or moves the contents of *right* into the `vector`.

Example

C++

```
// vector_operator_as.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> v1, v2, v3;
   vector<int>::iterator iter;

   v1.push_back(10);
   v1.push_back(20);
   v1.push_back(30);
   v1.push_back(40);
   v1.push_back(50);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
```

```
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
}
```

`pointer`

A type that provides a pointer to an element in a vector.

C++

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type `pointer` can be used to modify the value of an element.

Example

C++

```
// vector_pointer.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
    using namespace std;
    vector<int> v;
    v.push_back( 11 );
    v.push_back( 22 );

    vector<int>::pointer ptr = &v[0];
    cout << *ptr << endl;
    ptr++;
    cout << *ptr << endl;
    *ptr = 44;
    cout << *ptr << endl;
}
```

Output

```
11
22
```

```
44
```

pop_back

Deletes the element at the end of the vector.

C++

```
void pop_back();
```

Remarks

For a code example, see vector::push_back().

push_back

Adds an element to the end of the vector.

C++

```
void push_back(const T& value);

void push_back(T&& value);
```

Parameters

value

The value to assign to the element added to the end of the vector.

Example

C++

```
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}
```

```
template <typename T> void print_collection(const T& t) {
    cout << "  " << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(10 + i);
    }

    cout << "vector data: " << endl;
    print_collection(v);

    // pop_back() until it's empty, printing the last element as we go
    while (v.begin() != v.end()) {
        cout << "v.back(): "; print_elem(v.back()); cout << endl;
        v.pop_back();
    }
}
```

`rbegin`

Returns an iterator to the first element in a reversed vector.

C++

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Return value

A reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed vector.

Remarks

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the vector object can't be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the vector object can be modified.

Example

C++

```
// vector_rbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;
   vector <int>::reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   v1_Iter = v1.begin( );
   cout << "The first element of vector is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.rbegin( );
   cout << "The first element of the reversed vector is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

`reference`

A type that provides a reference to an element stored in a vector.

C++

```
typedef typename Allocator::reference reference;
```

Example

See at for an example of how to use `reference` in the vector class.

`rend`

Returns a past-the-end reverse iterator that points to the element following the last element of the reversed vector.

```
C++
```

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

Return value

A reverse past-the-end iterator for the reversed vector. It points to the element following the last element of the reversed vector, which is the same as the element before the first element of the non-reversed vector. That element is a placeholder and shouldn't be dereferenced. Only use it for comparisons.

Remarks

`rend` is used with a reversed vector just as end is used with a vector.

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the vector object can't be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the vector object can be modified.

`rend` can be used to test to whether a reverse iterator has reached the end of its vector.

The value returned by `rend` shouldn't be dereferenced. Only use it for comparisons.

Example

```
C++
```

```
// vector_rend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::reverse_iterator v1_rIter;
```

```
   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

reserve

Reserves a minimum length of storage for a vector object, allocating space if necessary.

C++

```
void reserve(size_type count);
```

Parameters

count

The minimum length of storage to be allocated for the vector.

Example

C++

```
// vector_reserve.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   //vector <int>::iterator Iter;

   v1.push_back( 1 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.reserve( 20 );
   cout << "Current capacity of v1 = "
```

```
      << v1.capacity( ) << endl;
}
```

Output

```
Current capacity of v1 = 1
Current capacity of v1 = 20
```

`resize`

Specifies a new size for a vector.

C++

```
void resize(size_type new_size);
void resize(size_type new_size, Type value);
```

Parameters

`new_size`

The new size of the vector.

`value`

The initialization value of new elements added to the vector if the new size is larger that the original size. If the value is omitted, the new objects use their default constructor.

Remarks

If the container's size is less than the requested size, *`new_size`*, `resize` adds elements to the vector until it reaches the requested size. When the container's size is larger than the requested size, `resize` deletes the elements closest to the end of the container until it reaches the size *`new_size`*. No action is taken if the present size of the container is the same as the requested size.

size reflects the current size of the vector.

Example

C++

```cpp
// vectorsizing.cpp
// compile with: /EHsc /W4
// Illustrates vector::reserve, vector::max_size,
// vector::resize, vector::resize, and vector::capacity.
//
// Functions:
//
//    vector::max_size - Returns maximum number of elements vector could
//                       hold.
//
//    vector::capacity - Returns number of elements for which memory has
//                       been allocated.
//
//    vector::size - Returns number of elements in the vector.
//
//    vector::resize - Reallocates memory for vector, preserves its
//                     contents if new size is larger than existing size.
//
//    vector::reserve - Allocates elements for vector to ensure a minimum
//                      size, preserving its contents if the new size is
//                      larger than existing size.
//
//    vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
////////////////////////////////////////////////////////////////////////

// The debugger cannot handle symbols more than 255 characters long.
// The C++ Standard Library often creates symbols longer than that.
// The warning can be disabled:
//#pragma warning(disable:4786)

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }
    cout << endl;
}

void printvstats(const vector<int>& v) {
    cout << "   the vector's size is: " << v.size() << endl;
    cout << "   the vector's capacity is: " << v.capacity() << endl;
    cout << "   the vector's maximum size is: " << v.max_size() << endl;
}

int main()
```

```
{
    // declare a vector that begins with 0 elements.
    vector<int> v;

    // Show statistics about vector.
    cout << endl << "After declaring an empty vector:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    // Add one element to the end of the vector.
    v.push_back(-1);
    cout << endl << "After adding an element:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }
    cout << endl << "After adding 10 elements:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    v.resize(6);
    cout << endl << "After resizing to 6 elements without an initialization
value:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    v.resize(9, 999);
    cout << endl << "After resizing to 9 elements with an initialization
value of 999:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    v.resize(12);
    cout << endl << "After resizing to 12 elements without an initialization
value:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    // Ensure there's room for at least 1000 elements.
    v.reserve(1000);
    cout << endl << "After vector::reserve(1000):" << endl;
    printvstats(v);

    // Ensure there's room for at least 2000 elements.
    v.resize(2000);
    cout << endl << "After vector::resize(2000):" << endl;
    printvstats(v);
}
```

reverse_iterator

A type that provides a random-access iterator that can read or modify any element in a reversed vector.

```
C++
```

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the vector in reverse.

Example

See the example for rbegin.

`shrink_to_fit`

Discards excess capacity.

```
C++
```

```
void shrink_to_fit();
```

Example

```
C++
```

```
// vector_shrink_to_fit.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   //vector <int>::iterator Iter;

   v1.push_back( 1 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.reserve( 20 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.shrink_to_fit();
```

```
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
}
```

Output

```
Current capacity of v1 = 1
Current capacity of v1 = 20
Current capacity of v1 = 1
```

`size`

Returns the number of elements in the vector.

C++

```
size_type size() const;
```

Return value

The current length of the vector.

Example

C++

```
// vector_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   v1.push_back( 1 );
   i = v1.size( );
   cout << "Vector length is " << i << "." << endl;

   v1.push_back( 2 );
   i = v1.size( );
   cout << "Vector length is now " << i << "." << endl;
}
```

Output

```
Vector length is 1.
Vector length is now 2.
```

`size_type`

A type that counts the number of elements in a vector.

C++

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for capacity.

`swap`

Exchanges the elements of two vectors.

C++

```
void swap(
    vector<Type, Allocator>& right);

friend void swap(
    vector<Type, Allocator>& left,
    vector<Type, Allocator>& right);
```

Parameters

`right`
A vector providing the elements to be swapped. Or, a vector whose elements are to be exchanged with the elements in the vector *`left`*.

`left`
A vector whose elements are to be exchanged with the elements in the vector *`right`*.

Example

C++

```
// vector_swap.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1, v2;

   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 3 );

   v2.push_back( 10 );
   v2.push_back( 20 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
   cout << endl;

   v1.swap( v2 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
}
```

Output

```
The number of elements in v1 = 3
The number of elements in v2 = 2

The number of elements in v1 = 2
The number of elements in v2 = 3
```

value_type

A type that represents the data type stored in a vector.

C++

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Type`.

Example

C++

```
// vector_value_type.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

`vector`

Constructs a vector. Overloads construct a vector of a specific size, or with elements of a specific value. Or, as a copy of all or part of some other vector. Some overloads also allow you to specify the allocator to use.

C++

```
vector();
explicit vector(const Allocator& allocator);
explicit vector(size_type count);
vector(size_type count, const Type& value);
vector(size_type count, const Type& value, const Allocator& allocator);

vector(const vector& source);
vector(vector&& source);
vector(initializer_list<Type> init_list, const Allocator& allocator);

template <class InputIterator>
vector(InputIterator first, InputIterator last);
template <class InputIterator>
vector(InputIterator first, InputIterator last, const Allocator& allocator);
```

Parameters

allocator

The allocator class to use with this object. get_allocator returns the allocator class for the object.

count

The number of elements in the constructed vector.

value

The value of the elements in the constructed vector.

source

The vector of which the constructed vector is to be a copy.

first

Position of the first element in the range of elements to be copied.

last

Position of the first element beyond the range of elements to be copied.

init_list

The `initializer_list` containing the elements to copy.

Remarks

All constructors store an allocator object (*allocator*) and initialize the vector.

The first two constructors specify an empty initial vector. The second constructor explicitly specifies the allocator type (*allocator*) to use.

The third constructor specifies a repetition of a specified number (*count*) of elements of the default value for class `Type`.

The fourth and fifth constructors specify a repetition of (*count*) elements of value *value*.

The sixth constructor specifies a copy of the vector *source*.

The seventh constructor moves the vector *source*.

The eighth constructor uses an initializer_list to specify the elements.

The ninth and tenth constructors copy the range [`first`, `last`) of a vector.

Example

C++

```
// vector_ctor.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector <int>::iterator v1_Iter, v2_Iter, v3_Iter, v4_Iter, v5_Iter,
v6_Iter;

    // Create an empty vector v0
    vector <int> v0;

    // Create a vector v1 with 3 elements of default value 0
    vector <int> v1(3);

    // Create a vector v2 with 5 elements of value 2
    vector <int> v2(5, 2);

    // Create a vector v3 with 3 elements of value 1 and with the allocator
    // of vector v2
    vector <int> v3(3, 1, v2.get_allocator());

    // Create a copy, vector v4, of vector v2
    vector <int> v4(v2);

    // Create a new temporary vector for demonstrating copying ranges
    vector <int> v5(5);
    for (auto i : v5) {
        v5[i] = i;
    }

    // Create a vector v6 by copying the range v5[ first,  last)
    vector <int> v6(v5.begin() + 1, v5.begin() + 3);

    cout << "v1 =";
    for (auto& v : v1){
        cout << " " << v;
    }
    cout << endl;

    cout << "v2 =";
    for (auto& v : v2){
        cout << " " << v;
    }
    cout << endl;

    cout << "v3 =";
    for (auto& v : v3){
```

```
        cout << " " << v;
    }
    cout << endl;
    cout << "v4 =";
    for (auto& v : v4){
        cout << " " << v;
    }
    cout << endl;

    cout << "v5 =";
    for (auto& v : v5){
        cout << " " << v;
    }
    cout << endl;

    cout << "v6 =";
    for (auto& v : v6){
        cout << " " << v;
    }
    cout << endl;

    // Move vector v2 to vector v7
    vector <int> v7(move(v2));
    vector <int>::iterator v7_Iter;

    cout << "v7 =";
    for (auto& v : v7){
        cout << " " << v;
    }
    cout << endl;

    cout << "v8 =";
    vector<int> v8{ { 1, 2, 3, 4 } };
    for (auto& v : v8){
        cout << " " << v ;
    }
    cout << endl;
}
```

Output

```
v1 = 0 0 0
v2 = 2 2 2 2 2
v3 = 1 1 1
v4 = 2 2 2 2 2
v5 = 0 0 0 0 0
v6 = 0 0
v7 = 2 2 2 2 2
v8 = 1 2 3 4
```

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`vector<bool>` Class

Article • 01/20/2023

The `vector<bool>` class is a partial specialization of vector for elements of type `bool`. It has an allocator for the underlying type that's used by the specialization, which provides space optimization by storing one `bool` value per bit.

Syntax

C++

```
template <class Allocator = allocator<bool>>
class vector<bool, Allocator>
```

Remarks

This class template specialization behaves like `vector`, except for the differences explained in this article.

Operations that deal with the `bool` type correspond to values in the container storage. `allocator_traits::construct` isn't used to construct these values.

Typedefs

Type name	Description
const_pointer	A typedef to a `const_iterator` that can serve as a constant pointer to a Boolean element of the `vector<bool>`.
const_reference	A typedef for `bool`. After initialization, it doesn't observe updates to the original value.
pointer	A typedef to an `iterator` that can serve as a pointer to a Boolean element of the `vector<bool>`.

Member functions

Member function	Description

Member function	Description
flip	Reverses all bits in the `vector<bool>`.
swap	Exchanges the elements of two `vector<bool>`s.
operator[]	Returns a simulated reference to the `vector<bool>` element at a specified position.
`at`	Functions the same as the unspecialized vector::at function, except that it uses the proxy class vector<bool>::reference. Also see operator[].
`front`	Functions the same as the unspecialized vector::front function, except that it uses the proxy class vector<bool>::reference. Also see operator[].
`back`	Functions the same as the unspecialized vector::back function, except that it uses the proxy class vector<bool>::reference. Also see operator[].

Proxy Class

Name	Description
vector<bool>::reference Class	A class that acts as a proxy to simulate `bool&` behavior, and whose objects can provide references to elements (single bits) within a `vector<bool>` object.

Requirements

Header: `<vector>`

Namespace: `std`

`vector<bool>::const_pointer`

A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the `vector<bool>` object.

```
C++

typedef const_iterator const_pointer;
```

`vector<bool>::const_reference`

A type that describes an object that can serve as a constant reference to a Boolean element of the sequence contained by the `vector<bool>` object.

```
C++
```

```
typedef bool const_reference;
```

Remarks

For more information and code examples, see vector<bool>::reference::operator=.

`vector<bool>::flip`

Reverses all bits in a `vector<bool>`.

```
C++
```

```
void flip();
```

Example

```
C++
```

```
// vector_bool_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha; // format output for subsequent code

    vector<bool> vb = { true, false, false, true, true };
    cout << "The vector is:" << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vb.flip();

    cout << "The flipped vector is:" << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
```

```
    cout << endl;
}
```

vector<bool>::operator[]

Returns a simulated reference to the `vector<bool>` element at a specified position.

```
C++
```

```
vector<bool>::reference operator[](size_type Pos);

vector&<bool&>::const_reference operator[](size_type Pos) const;
```

Parameters

Pos
The position of the `vector<bool>` element.

Return Value

A vector<bool>::reference or vector<bool>::const_reference object that contains the value of the indexed element.

If the position specified is greater than or equal to the size of the container, the result is undefined.

Remarks

If you compile with `_ITERATOR_DEBUG_LEVEL` set, a run-time error occurs if you attempt to access an element outside the bounds of the vector. For more information, see Checked Iterators.

Example

This code example shows the correct use of `vector<bool>::operator[]` and two common coding mistakes, which are commented out. These mistakes cause errors because the address of the `vector<bool>::reference` object that `vector<bool>::operator[]` returns can't be taken.

```
C++
```

```
// cl.exe /EHsc /nologo /W4 /MTd
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;
    vector<bool> vb;

    vb.push_back(true);
    vb.push_back(false);

    //    bool* pb = &vb[1]; // conversion error - do not use
    //    bool& refb = vb[1];   // conversion error - do not use
    bool hold = vb[1];
    cout << "The second element of vb is " << vb[1] << endl;
    cout << "The held value from the second element of vb is " << hold <<
endl;

    // Note this doesn't modify hold.
    vb[1] = true;
    cout << "The second element of vb is " << vb[1] << endl;
    cout << "The held value from the second element of vb is " << hold <<
endl;
}
```

Output

```
The second element of vb is false
The held value from the second element of vb is false
The second element of vb is true
The held value from the second element of vb is false
```

`vector<bool>::pointer`

A type that describes an object that can serve as a pointer to a Boolean element of the sequence contained by the `vector<bool>` object.

C++

```
typedef iterator pointer;
```

`vector<bool>::reference` Class

The `vector<bool>::reference` class is a proxy class provided by the vector<bool> Class to simulate `bool&`.

Remarks

A simulated reference is required because C++ doesn't natively allow direct references to bits. `vector<bool>` uses only one bit per element, which can be referenced by using this proxy class. However, the reference simulation isn't complete because certain assignments aren't valid. For example, because the address of the `vector<bool>::reference` object can't be taken, the following code that uses vector<bool>::operator[] isn't correct:

```
C++
```

```
vector<bool> vb;
//...
bool* pb = &vb[1]; // conversion error - do not use
bool& refb = vb[1];   // conversion error - do not use
```

`vector<bool>::reference::flip`

Inverts the Boolean value of a referenced vector<bool> element.

```
C++
```

```
void flip();
```

Example

```
C++
```

```
// vector_bool_ref_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    cout << "The vector is: " << endl << "    ";
```

```
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vector<bool>::reference vbref = vb.front();
    vbref.flip();

    cout << "The vector with first element flipped is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;
}
```

Output

```
The vector is:
    true false false true true
The vector with first element flipped is:
    false false false true true
```

`vector<bool>::reference::operator bool`

Provides an implicit conversion from `vector<bool>::reference` to `bool`.

C++

```
operator bool() const;
```

Return Value

The Boolean value of the element of the `vector<bool>` object.

Remarks

The `vector<bool>` object can't be modified by this operator.

`vector<bool>::reference::operator=`

Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

C++

```
reference& operator=(const reference& Right);
reference& operator=(bool Val);
```

Parameters

Right

The element reference whose value is to be assigned to the bit.

Val

The Boolean value to be assigned to the bit.

Example

C++

```
// vector_bool_ref_op_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    print("The vector is: ", vb);

    // Invoke vector<bool>::reference::operator=()
    vector<bool>::reference refelem1 = vb[0];
    vector<bool>::reference refelem2 = vb[1];
    vector<bool>::reference refelem3 = vb[2];

    bool b1 = refelem1;
    bool b2 = refelem2;
    bool b3 = refelem3;
```

```
    cout << "The original value of the 1st element stored in a bool: " << b1
<< endl;
    cout << "The original value of the 2nd element stored in a bool: " << b2
<< endl;
    cout << "The original value of the 3rd element stored in a bool: " << b3
<< endl;
    cout << endl;

    refelem2 = refelem1;

    print("The vector after assigning refelem1 to refelem2 is now: ", vb);

    refelem3 = true;

    print("The vector after assigning false to refelem1 is now: ", vb);

    // The initial values are still stored in the bool variables and
remained unchanged
    cout << "The original value of the 1st element still stored in a bool: "
<< b1 << endl;
    cout << "The original value of the 2nd element still stored in a bool: "
<< b2 << endl;
    cout << "The original value of the 3rd element still stored in a bool: "
<< b3 << endl;
    cout << endl;
}
```

Output

```
The vector is: true false false true true
The original value of the 1st element stored in a bool: true
The original value of the 2nd element stored in a bool: false
The original value of the 3rd element stored in a bool: false

The vector after assigning refelem1 to refelem2 is now: true true false true
true
The vector after assigning false to refelem1 is now: true true true true
true
The original value of the 1st element still stored in a bool: true
The original value of the 2nd element still stored in a bool: false
The original value of the 3rd element still stored in a bool: false
```

`vector<bool>::swap`

Static member function that exchanges two elements of Boolean vectors (`vector<bool>`) by using the proxy class vector<bool>::reference.

C++

```
static void swap(
    reference Left,
    reference Right);
```

Parameters

Left

The element to be exchanged with the *Right* element.

Right

The element to be exchanged with the *Left* element.

Remarks

This overload supports the special proxy requirements of `vector<bool>`. vector::swap has the same functionality as the single-argument overload of `vector<bool>::swap()`.

See also

Thread Safety in the C++ Standard Library
C++ Standard Library Reference

`vector<bool>::reference` Class

Article • 12/06/2021

The `vector<bool>::reference` class is a proxy class provided by the vector<bool> Class to simulate `bool&`.

Remarks

A simulated reference is required because C++ does not natively allow direct references to bits. `vector<bool>` uses only one bit per element, which can be referenced by using this proxy class. However, the reference simulation is not complete because certain assignments are not valid. For example, because the address of the `vector<bool>::reference` object cannot be taken, the following code that tries to use `vector<bool>::operator&` is not correct:

```
C++
```

```
vector<bool> vb;
// ...
bool* pb = &vb[1]; // conversion error - do not use
bool& refb = vb[1];   // conversion error - do not use
```

Member functions

Member function	Description
flip	Inverts the Boolean value of a vector element.
operator bool	Provides an implicit conversion from `vector<bool>::reference` to `bool`.
operator=	Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

Requirements

Header: <vector>

Namespace: std

See also

<vector>

Thread Safety in the C++ Standard Library

C++ Standard Library Reference

vector<bool>::reference::flip

Article • 12/06/2021

Inverts the Boolean value of a referenced vector<bool> element.

Syntax

C++

```
void flip();
```

Example

C++

```
// vector_bool_ref_flip.cpp
// compile with: /EHsc /W4
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    cout << "The vector is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;

    vector<bool>::reference vbref = vb.front();
    vbref.flip();

    cout << "The vector with first element flipped is: " << endl << "    ";
    for (const auto& b : vb) {
        cout << b << " ";
    }
    cout << endl;
}
```

Output

Output

```
The vector is:
    true false false true true
The vector with first element flipped is:
    false false false true true
```

Requirements

Header: <vector>

Namespace: std

See also

vector<bool>::reference Class

C++ Standard Library Reference

`vector<bool>::reference::operator bool`

Article • 12/06/2021

Provides an implicit conversion from `vector<bool>::reference` to `bool`.

Syntax

```
C++

operator bool() const;
```

Return Value

The Boolean value of the element of the vector<bool> object.

Remarks

The `vector<bool>` object cannot be modified by this operator.

Requirements

Header: <vector>

Namespace: std

See also

vector<bool>::reference Class

C++ Standard Library Reference

vector<bool>::reference::operator=

Article • 12/06/2021

Assigns a Boolean value to a bit, or the value held by a referenced element to a bit.

Syntax

C++

```
reference& operator=(const reference& Right);

reference& operator=(bool Val);
```

Parameters

Right
The element reference whose value is to be assigned to the bit.

Val
The Boolean value to be assigned to the bit.

Example

C++

```
// vector_bool_ref_op_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
```

```
{
    cout << boolalpha;

    vector<bool> vb = { true, false, false, true, true };

    print("The vector is: ", vb);

    // Invoke vector<bool>::reference::operator=()
    vector<bool>::reference refelem1 = vb[0];
    vector<bool>::reference refelem2 = vb[1];
    vector<bool>::reference refelem3 = vb[2];

    bool b1 = refelem1;
    bool b2 = refelem2;
    bool b3 = refelem3;
    cout << "The original value of the 1st element stored in a bool: " << b1
<< endl;
    cout << "The original value of the 2nd element stored in a bool: " << b2
<< endl;
    cout << "The original value of the 3rd element stored in a bool: " << b3
<< endl;
    cout << endl;

    refelem2 = refelem1;

    print("The vector after assigning refelem1 to refelem2 is now: ", vb);

    refelem3 = true;

    print("The vector after assigning false to refelem1 is now: ", vb);

    // The initial values are still stored in the bool variables and
remained unchanged
    cout << "The original value of the 1st element still stored in a bool: "
<< b1 << endl;
    cout << "The original value of the 2nd element still stored in a bool: "
<< b2 << endl;
    cout << "The original value of the 3rd element still stored in a bool: "
<< b3 << endl;
    cout << endl;
}
```

Output

Output

```
The vector is: true false false true true
The original value of the 1st element stored in a bool: true
The original value of the 2nd element stored in a bool: false
The original value of the 3rd element stored in a bool: false

The vector after assigning refelem1 to refelem2 is now: true true false true
```

```
true
The vector after assigning false to refelem1 is now: true true true true
true
The original value of the 1st element still stored in a bool: true
The original value of the 2nd element still stored in a bool: false
The original value of the 3rd element still stored in a bool: false
```

Requirements

Header: <vector>

Namespace: std

See also

vector<bool>::reference Class

C++ Standard Library Reference

`<vector>` functions

Article • 03/04/2022

`swap`

Exchanges the elements of two vectors.

```
C++
```

```
template <class Type, class Allocator>
void swap(vector<Type, Allocator>& left, vector<Type, Allocator>& right);
```

Parameters

`right`
The vector providing the elements to be swapped, or the vector whose elements are to be exchanged with those of the vector *`left`*.

`left`
The vector whose elements are to be exchanged with those of the vector *`right`*.

Remarks

The template function is an algorithm specialized on the container class vector to execute the member function `left. vector::swap (right)`. These are instances of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call isn't unique, then the compiler will select the most specialized version of the template function. The general version of the template function, **`template <class T> void swap(T&, T&)`**, in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

Example

See the code example for member function vector::swap for an example that uses the template version of `swap`.

`<vector>` operators

Article • 12/07/2021

operator!=

Tests if the object on the left side of the operator is not equal to the object on the right side.

C++

```
bool operator!=(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vectors are not equal; `false` if the vectors are equal.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// vector_op_ne.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
```

```
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
     v2.push_back( 2 );

   if ( v1 != v2 )
      cout << "Vectors not equal." << endl;
   else
      cout << "Vectors equal." << endl;
}
```

Output

```
Vectors not equal.
```

`operator<`

Tests if the object on the left side of the operator is less than the object on the right side.

C++

```
bool operator<(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vector on the left side of the operator is less than the vector on the right side of the operator; otherwise **`false`**.

Example

C++

```cpp
// vector_op_lt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 4 );

   v2.push_back( 1 );
   v2.push_back( 3 );

   if ( v1 < v2 )
      cout << "Vector v1 is less than vector v2." << endl;
   else
      cout << "Vector v1 is not less than vector v2." << endl;
}
```

Output

```
Vector v1 is less than vector v2.
```

`operator<=`

Tests if the object on the left side of the operator is less than or equal to the object on the right side.

C++

```cpp
bool operator<=(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vector on the left side of the operator is less than or equal to the vector on the right side of the operator; otherwise `false`.

Example

C++

```
// vector_op_le.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 4 );

   v2.push_back( 1 );
   v2.push_back( 3 );

   if ( v1 <= v2 )
      cout << "Vector v1 is less than or equal to vector v2." << endl;
   else
      cout << "Vector v1 is greater than vector v2." << endl;
}
```

Output

```
Vector v1 is less than or equal to vector v2.
```

operator==

Tests if the object on the left side of the operator is equal to the object on the right side.

C++

```
bool operator==(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vector on the left side of the operator is equal to the vector on the right side of the operator; otherwise `false`.

Remarks

Two vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

Example

C++

```
// vector_op_eq.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v2.push_back( 1 );

   if ( v1 == v2 )
      cout << "Vectors equal." << endl;
   else
      cout << "Vectors not equal." << endl;
}
```

Output

```
Vectors equal.
```

`operator>`

Tests if the object on the left side of the operator is greater than the object on the right side.

```
C++
```

```
bool operator>(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vector on the left side of the operator is greater than the vector on the right side of the operator; otherwise **`false`**.

Example

```
C++
```

```
// vector_op_gt.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 3 );
   v1.push_back( 1 );

   v2.push_back( 1 );
   v2.push_back( 2 );
   v2.push_back( 2 );
```

```
   if ( v1 > v2 )
      cout << "Vector v1 is greater than vector v2." << endl;
   else
      cout << "Vector v1 is not greater than vector v2." << endl;
}
```

Output

```
Vector v1 is greater than vector v2.
```

`operator>=`

Tests if the object on the left side of the operator is greater than or equal to the object on the right side.

C++

```
bool operator>=(const vector<Type, Allocator>& left, const vector<Type,
Allocator>& right);
```

Parameters

left
An object of type `vector`.

right
An object of type `vector`.

Return Value

`true` if the vector on the left side of the operator is greater than or equal to the vector on the right side of the vector; otherwise **`false`**.

Example

C++

```
// vector_op_ge.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
```

```
int main( )
{
   using namespace std;

   vector <int> v1, v2;
   v1.push_back( 1 );
   v1.push_back( 3 );
   v1.push_back( 1 );

     v2.push_back( 1 );
   v2.push_back( 2 );
   v2.push_back( 2 );

   if ( v1 >= v2 )
      cout << "Vector v1 is greater than or equal to vector v2." << endl;
   else
      cout << "Vector v1 is less than vector v2." << endl;
}
```

Output

```
Vector v1 is greater than or equal to vector v2.
```

C++ Standard Library (STL) overview

Article • 09/22/2022

All C++ library entities are declared or defined in one or more standard headers. This implementation includes two other headers, `<hash_map>` and `<hash_set>`, that aren't required by the C++ Standard. For a complete list of headers that this implementation supports, see Header files reference.

The C++ standard defines two kinds of conforming libraries:

- A *hosted implementation*, which supports all of the required standard library headers described by the C++ ISO standard.
- A *freestanding implementation*, which requires only a subset of the standard library headers. The required subset is:

Freestanding header subset		
<atomic> (declaring at least `atomic_signed_lock_free` and `atomic_unsigned_lock_free`)	<cstdint>	<ranges>
<bit>	<cstdlib> (declaring at least `abort`, `atexit`, `at_quick_exit`, `exit`, `quick_exit`)	<ratio>
<cfloat>	<exception>	<tuple>
<climits>	<functional>	<typeinfo>
`<compare>`	<initializer_list>	`<source_location>`
`<concepts>`	<iterator>	<type_traits>
`<coroutine>`	<limits>	<utility>
<cstdarg>	<memory>	`<version>`
<cstddef>	<new>	

The following headers are deprecated since C++11: <ciso646>, <cstdalign>, and <cstdbool>.

Other differences between freestanding and hosted implementations are:

- Hosted implementations require a global function named `main`. A freestanding implementation can define its own startup and termination functions.

- Hosted implementations must support more than one thread running at the same time. Implementors of freestanding implementations decide whether their library supports concurrent threads.

The Microsoft C++ standard library satisfies both freestanding and hosted requirements.

The C++ library headers have two broader subdivisions:

- iostreams conventions.
- C++ Standard library (STL) reference conventions.

This section contains the following sections:

- Using C++ library headers
- C++ library conventions
- iostreams Conventions
- C++ program startup and termination
- Safe libraries: C++ standard library
- Checked iterators
- Debug iterator support
- C++ standard library (STL) reference
- Thread safety in the C++ standard library
- stdext namespace
- Regular expressions (C++)

For more information about Visual C++ run-time libraries, see CRT Library Features.

> **Note**
>
> Microsoft's implementation of the C++ Standard Library is often referred to as the *STL* or *Standard Template Library*. Although *C++ Standard Library* is the official name of the library as defined in ISO 14882, due to the popular use of "STL" and "Standard Template Library" in search engines, we occasionally use those names to make it easier to find our documentation. From a historical perspective, "STL" originally referred to the Standard Template Library written by Alexander Stepanov. Parts of that library were standardized in the C++ Standard Library, along with the

ISO C runtime library, parts of the Boost library, and other functionality. Sometimes "STL" is also used to refer to the containers and algorithms parts of the C++ Standard Library adapted from Stepanov's STL. In this documentation, Standard Template Library (STL) refers to the C++ Standard Library as a whole.

See also

C++ Standard Library

Using C++ Library Headers

Article • 02/08/2022

You include the contents of a standard header by naming it in an include directive.

```
C++

#include <iostream>// include I/O facilities
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Don't include a standard header within a declaration. Don't define macros that have the same names as keywords before you include a standard header.

A C++ library header includes any other C++ library headers it needs to define needed types. (Always include explicitly any C++ library headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro with the same name as the function that masks the function declaration and achieves the same effect. For more information on masking macros, see C++ Library Conventions.

All names other than `operator delete` and `operator new` in the C++ library headers are defined in the `std` namespace, or in a namespace nested within the `std` namespace. You refer to the name `cin`, for example, as `std::cin`. Note, however, that macro names aren't subject to namespace qualification, so you always write `__STD_COMPLEX` without a namespace qualifier.

In some translation environments, including a C++ library header may hoist external names declared in the `std` namespace into the global namespace as well, with individual `using` declarations for each of the names. Otherwise, the header doesn't* introduce any library names into the current namespace.

The C++ Standard requires that the C Standard headers declare all external names in namespace `std`, then hoist them into the global namespace with individual `using` declarations for each of the names. But in some translation environments the C Standard headers include no namespace declarations, declaring all names directly in the global namespace. Thus, the most portable way to deal with namespaces is to follow two rules:

- To assuredly declare in namespace `std` an external name that is traditionally declared in `<stdlib.h>`, for example, include the header `<cstdlib>`. Know that the name might also be declared in the global namespace.
- To assuredly declare in the global namespace an external name declared in `<stdlib.h>`, include the header `<stdlib.h>` directly. Know that the name might also be declared in namespace `std`.

Thus, if you want to call `std::abort` to cause abnormal termination, you should include `<cstdlib>`. If you want to call `abort`, you should include `<stdlib.h>`.

Alternatively, you can write the declaration:

```
C++

using namespace std;
```

which brings all library names into the current namespace. If you write this declaration immediately after all include directives, you hoist the names into the global namespace. You can subsequently ignore namespace considerations in the remainder of the translation unit. You also avoid most differences across different translation environments.

Unless specifically indicated otherwise, you may not define names in the `std` namespace, or in a namespace nested within the `std` namespace, within your program.

See also

C++ Standard Library Overview
Thread Safety in the C++ Standard Library

C++ Library Conventions

Article • 08/03/2021

The C++ library obeys much the same conventions as the Standard C Library, plus a few more outlined here.

An implementation has certain latitude in how it declares types and functions in the C++ library:

- Names of functions in the Standard C library may have either extern "C++" or extern "C" linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (nonvirtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different integer types.
- A bitmask type can be implemented as either an integer type or an enumeration. In either case, you can perform bitwise operations (such as `AND` and `OR`) on values of the same bitmask type. The elements `A` and `B` of a bitmask type are nonzero values such that `A` & `B` is zero.
- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions:

- The Standard C Library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.
- A library function name outside a class will not have additional, undocumented, function signatures. You can reliably take its address.
- Base classes and member functions described as virtual are assuredly virtual, while those described as nonvirtual are assuredly nonvirtual.

- Two types defined by the C++ library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions, can throw *at most* those exceptions listed in any exception specification. No destructors supplied by the library throw exceptions. Functions in the Standard C Library may propagate an exception, as when `qsort` calls a comparison function that throws an exception, but they do not otherwise throw exceptions.

See also

C++ Standard Library Overview
Thread Safety in the C++ Standard Library

C++ Program Startup and Termination

Article • 08/03/2021

A C++ program performs the same operations as a C program does at program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function `main`, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that some iostreams objects are properly initialized for use by these static constructors. These control text streams are:

- cin — for standard input.
- cout — for standard output.
- cerr — for unbuffered standard error output.
- clog — for buffered standard error output.

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from `main` or calling `exit` calls all functions registered with `atexit` in reverse order of registry. An exception thrown from such a registered function calls `terminate`.

See also

C++ Standard Library Overview
Thread Safety in the C++ Standard Library

Safe Libraries: C++ Standard Library

Article • 08/03/2021

Several enhancements have been made to the libraries that ship with Microsoft C++, including the C++ Standard Library, to make them more secure.

Several methods in the C++ Standard Library have been identified as potentially unsafe because they could lead to a buffer overrun or other code defect. The use of these methods is discouraged, and new, more secure methods have been created to replace them. These new methods all end in `_s`.

Several enhancements have also been made to make iterators and algorithms more secure. For more information, see Checked Iterators, Debug Iterator Support and _ITERATOR_DEBUG_LEVEL.

Remarks

The following table lists the C++ Standard Library methods that are potentially unsafe, as well as their safer equivalent:

Potentially unsafe method	Safer equivalent
copy	basic_string::_Copy_s
copy	char_traits::_Copy_s

If you call any one of the potentially unsafe methods above, or if you use iterators incorrectly, the compiler will generate Compiler Warning (level 3) C4996. For information on how to disable these warnings, see _SCL_SECURE_NO_WARNINGS.

In This Section

_ITERATOR_DEBUG_LEVEL

_SCL_SECURE_NO_WARNINGS

Checked Iterators

Debug Iterator Support

See also

C++ Standard Library Overview

_ITERATOR_DEBUG_LEVEL

Article • 08/03/2021

The `_ITERATOR_DEBUG_LEVEL` macro controls whether checked iterators and debug iterator support are enabled. This macro supersedes and combines the functionality of the older `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` macros.

Macro Values

The following table summarizes the possible values for the `_ITERATOR_DEBUG_LEVEL` macro.

Compilation mode	Macro value	Description
Debug		
	0	Disables checked iterators and disables iterator debugging.
	1	Enables checked iterators and disables iterator debugging.
	2 (Default)	Enables iterator debugging; checked iterators are not relevant.
Release		
	0 (Default)	Disables checked iterators.
	1	Enables checked iterators; iterator debugging is not relevant.

In release mode, the compiler generates an error if you specify `_ITERATOR_DEBUG_LEVEL` as 2.

Remarks

The `_ITERATOR_DEBUG_LEVEL` macro controls whether checked iterators are enabled, and in Debug mode, whether debug iterator support is enabled. If `_ITERATOR_DEBUG_LEVEL` is defined as 1 or 2, checked iterators ensure that the bounds of your containers are not overwritten. If `_ITERATOR_DEBUG_LEVEL` is 0, iterators are not checked. When `_ITERATOR_DEBUG_LEVEL` is defined as 1, any unsafe iterator use causes a runtime error and the program is terminated. When `_ITERATOR_DEBUG_LEVEL` is defined as 2, unsafe

iterator use causes an assert and a runtime error dialog that lets you break into the debugger.

Because the `_ITERATOR_DEBUG_LEVEL` macro supports similar functionality to the `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` macros, you may be uncertain which macro and macro value to use in a particular situation. To prevent confusion, we recommend that you use only the `_ITERATOR_DEBUG_LEVEL` macro. This table describes the equivalent `_ITERATOR_DEBUG_LEVEL` macro value to use for various values of `_SECURE_SCL` and `_HAS_ITERATOR_DEBUGGING` in existing code.

`_ITERATOR_DEBUG_LEVEL`	`_SECURE_SCL`	`_HAS_ITERATOR_DEBUGGING`
0 (Release default)	0 (disabled)	0 (disabled)
1	1 (enabled)	0 (disabled)
2 (Debug default)	(not relevant)	1 (enabled in Debug mode)

For information on how to disable warnings about checked iterators, see _SCL_SECURE_NO_WARNINGS.

Example

To specify a value for the `_ITERATOR_DEBUG_LEVEL` macro, use a /D compiler option to define it on the command line, or use `#define` before the C++ Standard Library headers are included in your source files. For example, on the command line, to compile *sample.cpp* in debug mode and to use debug iterator support, you can specify the `_ITERATOR_DEBUG_LEVEL` macro definition:

```
cl /EHsc /Zi /MDd /D_ITERATOR_DEBUG_LEVEL=1 sample.cpp
```

In a source file, specify the macro before any standard library headers that define iterators.

C++

```
// sample.cpp

#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>

// ...
```

See also

Checked Iterators

Debug Iterator Support

Safe Libraries: C++ Standard Library

_SCL_SECURE_NO_WARNINGS

Article • 08/03/2021

Calling any of the potentially unsafe methods in the C++ Standard Library results in Compiler Warning (level 3) C4996. To disable this warning, define the macro _SCL_SECURE_NO_WARNINGS in your code:

```
C++

#define _SCL_SECURE_NO_WARNINGS
```

If you use precompiled headers, put this directive in your precompiled header file before you include any C runtime library or standard library headers. If you put it in an individual source code file before you include the precompiled header file, it is ignored by the compiler.

Remarks

Other ways to disable warning C4996 include:

- Using the /D (Preprocessor Definitions) compiler option:

  > cl /D_SCL_SECURE_NO_WARNINGS [other compiler options] myfile.cpp

- Using the /w compiler option:

  > cl /wd4996 [other compiler options] myfile.cpp

- Using the #pragma warning directive:

  ```
  C++

  #pragma warning(disable:4996)
  ```

Also, you can manually change the level of warning C4996 with the **/w<l><n>** compiler option. For example, to set warning C4996 to level 4:

> cl /w44996 [other compiler options] myfile.cpp

For more information, see /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning Level).

See also

Safe Libraries: C++ Standard Library

Checked Iterators

Article • 08/03/2021

Checked iterators ensure that the bounds of your container are not overwritten. Checked iterators apply to both release builds and debug builds. For more information about how to use debug iterators when you compile in debug mode, see Debug Iterator Support.

Remarks

For information about how to disable warnings that are generated by checked iterators, see _SCL_SECURE_NO_WARNINGS.

You can use the _ITERATOR_DEBUG_LEVEL preprocessor macro to enable or disable the checked iterators feature. If _ITERATOR_DEBUG_LEVEL is defined as 1 or 2, unsafe use of iterators causes a runtime error and the program is terminated. If defined as 0, checked iterators are disabled. By default, the value for _ITERATOR_DEBUG_LEVEL is 0 for release builds and 2 for debug builds.

> **ⓘ Important**
>
> Older documentation and source code may refer to the **_SECURE_SCL** macro. Use _ITERATOR_DEBUG_LEVEL to control _SECURE_SCL. For more information, see **_ITERATOR_DEBUG_LEVEL**.

When _ITERATOR_DEBUG_LEVEL is defined as 1 or 2, these iterator checks are performed:

- All standard iterators (for example, vector::iterator) are checked.
- If an output iterator is a checked iterator, calls to standard library functions such as std::copy get checked behavior.
- If an output iterator is an unchecked iterator, calls to standard library functions cause compiler warnings.
- The following functions generate a runtime error if there is an access that is outside the bounds of the container:

  basic_string::operator[]
  bitset::operator[]

deque::back
deque::front

deque::operator[]
list::back
list::front
queue::back

queue::front
vector::back
vector::front
vector::operator[]

When _ITERATOR_DEBUG_LEVEL is defined as 0:

- All standard iterators are unchecked. Iterators can move beyond the container boundaries, which leads to undefined behavior.
- If an output iterator is a checked iterator, calls to standard library functions such as `std::copy` get checked behavior.
- If an output iterator is an unchecked iterator, calls to standard library functions get unchecked behavior.

A checked iterator refers to an iterator that calls `invalid_parameter_handler` if you attempt to move past the boundaries of the container. For more information about `invalid_parameter_handler`, see Parameter Validation.

The iterator adaptors that support checked iterators are checked_array_iterator Class and unchecked_array_iterator Class.

Examples

When you compile by using _ITERATOR_DEBUG_LEVEL set to 1 or 2, a runtime error will occur if you attempt to access an element that is outside the bounds of the container by using the indexing operator of certain classes.

C++

```
// checked_iterators_1.cpp
// cl.exe /Zi /MDd /EHsc /W4

#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>
```

```
#include <iostream>

using namespace std;

int main()
{
   vector<int> v;
   v.push_back(67);

   int i = v[0];
   cout << i << endl;

   i = v[1]; //triggers invalid parameter handler
}
```

This program prints "67" then pops up an assertion failure dialog box with additional information about the failure.

Similarly, when you compile by using _ITERATOR_DEBUG_LEVEL set to 1 or 2, a runtime error will occur if you attempt to access an element by using `front` or `back` in container classes when the container is empty.

C++

```
// checked_iterators_2.cpp
// cl.exe /Zi /MDd /EHsc /W4
#define _ITERATOR_DEBUG_LEVEL 1

#include <vector>
#include <iostream>

using namespace std;

int main()
{
   vector<int> v;

   int& i = v.front(); // triggers invalid parameter handler
}
```

This program pops up an assertion failure dialog box with additional information about the failure.

The following code demonstrates various iterator use-case scenarios with comments about each. By default, _ITERATOR_DEBUG_LEVEL is set to 2 in Debug builds, and to 0 in Retail builds.

C++

```
// checked_iterators_3.cpp
// cl.exe /MTd /EHsc /W4

#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <numeric>
#include <string>
#include <vector>

using namespace std;

template <typename C>
void print(const string& s, const C& c)
{
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

int main()
{
    vector<int> v(16);
    iota(v.begin(), v.end(), 0);
    print("v: ", v);

    // OK: vector::iterator is checked in debug mode
    // (i.e. an overrun causes a debug assertion)
    vector<int> v2(16);
    transform(v.begin(), v.end(), v2.begin(), [](int n) { return n * 2; });
    print("v2: ", v2);

    // OK: back_insert_iterator is marked as checked in debug mode
    // (i.e. an overrun is impossible)
    vector<int> v3;
    transform(v.begin(), v.end(), back_inserter(v3), [](int n) { return n *
3; });
    print("v3: ", v3);

    // OK: array::iterator is checked in debug mode
    // (i.e. an overrun causes a debug assertion)
    array<int, 16> a4;
    transform(v.begin(), v.end(), a4.begin(), [](int n) { return n * 4; });
    print("a4: ", a4);

    // OK: Raw arrays are checked in debug mode
    // (an overrun causes a debug assertion)
    // NOTE: This applies only when raw arrays are given to C++ Standard
Library algorithms!
```

```
    int a5[16];
    transform(v.begin(), v.end(), a5, [](int n) { return n * 5; });
    print("a5: ", a5);

    // WARNING C4996: Pointers cannot be checked in debug mode
    // (an overrun causes undefined behavior)
    int a6[16];
    int * p6 = a6;
    transform(v.begin(), v.end(), p6, [](int n) { return n * 6; });
    print("a6: ", a6);

    // OK: stdext::checked_array_iterator is checked in debug mode
    // (an overrun causes a debug assertion)
    int a7[16];
    int * p7 = a7;
    transform(v.begin(), v.end(), stdext::make_checked_array_iterator(p7,
16), [](int n) { return n * 7; });
    print("a7: ", a7);

    // WARNING SILENCED: stdext::unchecked_array_iterator is marked as
checked in debug mode
    // (it performs no checking, so an overrun causes undefined behavior)
    int a8[16];
    int * p8 = a8;
    transform(v.begin(), v.end(), stdext::make_unchecked_array_iterator(p8),
[](int n) { return n * 8; });
    print("a8: ", a8);
}
```

When you compile this code by using `cl.exe /EHsc /W4 /MTd checked_iterators_3.cpp` the compiler emits a warning, but compiles without error into an executable:

Output

```
algorithm(1026) : warning C4996: 'std::_Transform1': Function call with
parameters
that may be unsafe - this call relies on the caller to check that the passed
values
are correct. To disable this warning, use -D_SCL_SECURE_NO_WARNINGS. See
documentation
on how to use Visual C++ 'Checked Iterators'
```

When run at the command line, the executable generates this output:

Output

```
v: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
v2: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
v3: 0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
a4: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
a5: 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
```

```
a6: 0 6 12 18 24 30 36 42 48 54 60 66 72 78 84 90
a7: 0 7 14 21 28 35 42 49 56 63 70 77 84 91 98 105
a8: 0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
```

See also

C++ Standard Library Overview

Debug Iterator Support

_SECURE_SCL

Article • 08/03/2021

Superseded by _ITERATOR_DEBUG_LEVEL, this macro defines whether Checked Iterators are enabled. By default, checked iterators are enabled in Debug builds, and disabled in Retail builds.

> **ⓘ Important**
>
> Direct use of the _SECURE_SCL macro is deprecated. Instead, use _ITERATOR_DEBUG_LEVEL to control checked iterator settings. For more information, see **_ITERATOR_DEBUG_LEVEL**.

Remarks

When checked iterators are enabled, unsafe iterator use causes a runtime error and the program is terminated. To enable checked iterators, set _ITERATOR_DEBUG_LEVEL to 1 or 2. This is equivalent to a _SECURE_SCL setting of 1, or enabled:

```
C++

#define _ITERATOR_DEBUG_LEVEL 1
```

To disable checked iterators, set _ITERATOR_DEBUG_LEVEL to 0. This is equivalent to a _SECURE_SCL setting of 0, or disabled:

```
C++

#define _ITERATOR_DEBUG_LEVEL 0
```

For information on how to disable warnings about checked iterators, see _SCL_SECURE_NO_WARNINGS.

See also

_ITERATOR_DEBUG_LEVEL
Checked Iterators
Debug Iterator Support
Safe Libraries: C++ Standard Library

Debug Iterator Support

Article • 08/15/2022

The Visual C++ run-time library detects incorrect iterator use, and asserts and displays a dialog box at run time. To enable debug iterator support, you must use debug versions of the C++ Standard Library and C Runtime Library to compile your program. For more information, see CRT Library Features. For information about how to use checked iterators, see Checked Iterators.

The C++ standard describes how member functions might cause iterators to a container to become invalid. Two examples are:

- Erasing an element from a container causes iterators to the element to become invalid.
- Increasing the size of a vector by using push or insert causes iterators into the `vector` to become invalid.

Invalid iterators

If you compile this sample program in debug mode, at run time it asserts and terminates.

C++

```
// iterator_debugging_0.cpp
// compile by using /EHsc /MDd
#include <vector>
#include <iostream>

int main() {
   std::vector<int> v {10, 15, 20};
   std::vector<int>::iterator i = v.begin();
   ++i;

   std::vector<int>::iterator j = v.end();
   --j;

   std::cout << *j << '\n';

   v.insert(i,25);

   std::cout << *j << '\n'; // Using an old iterator after an insert
}
```

Using _ITERATOR_DEBUG_LEVEL

You can use the preprocessor macro _ITERATOR_DEBUG_LEVEL to turn off the iterator debugging feature in a debug build. This program doesn't assert, but still triggers undefined behavior.

C++

```
// iterator_debugging_1.cpp
// compile by using: /EHsc /MDd
#define _ITERATOR_DEBUG_LEVEL 0
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v {10, 15, 20};

   std::vector<int>::iterator i = v.begin();
   ++i;

   std::vector<int>::iterator j = v.end();
   --j;

   std::cout << *j << '\n';

   v.insert(i,25);

   std::cout << *j << '\n'; // Using an old iterator after an insert
}
```

Output

```
20
-572662307
```

Uninitialized iterators

An assert also occurs if you attempt to use an iterator before it's initialized, as shown here:

C++

```
// iterator_debugging_2.cpp
// compile by using: /EHsc /MDd
#include <string>
using namespace std;

int main() {
```

```
    string::iterator i1, i2;
    if (i1 == i2)
        ;
}
```

Incompatible iterators

The following code example causes an assertion because the two iterators to the for_each algorithm are incompatible. Algorithms check to determine whether the iterators that are supplied to them reference the same container.

C++

```
// iterator_debugging_3.cpp
// compile by using /EHsc /MDd
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1 {10, 20};
    vector<int> v2 {10, 20};

    // The next line asserts because v1 and v2 are
    // incompatible.
    for_each(v1.begin(), v2.end(), [] (int& elem) { elem *= 2; } );
}
```

Notice that this example uses the lambda expression `[] (int& elem) { elem *= 2; }` instead of a functor. Although this choice has no bearing on the assert failure—a similar functor would cause the same failure—lambdas are a way to write a short block of code. For more information about lambda expressions, see Lambda expressions.

Iterators going out of scope

Debug iterator checks also cause an iterator variable that's declared in a `for` loop to be out of scope when the `for` loop scope ends.

C++

```
// iterator_debugging_4.cpp
// compile by using: /EHsc /MDd
#include <vector>
#include <iostream>
int main() {
```

```
   std::vector<int> v {10, 15, 20};

   for (std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
      ;   // do nothing
   --i;   // C2065
}
```

Destructors for debug iterators

Debug iterators have non-trivial destructors. If a destructor doesn't run but the object's memory is freed, access violations and data corruption might occur. Consider this example:

C++

```
// iterator_debugging_5.cpp
// compile by using: /EHsc /MDd
#include <vector>
struct base {
   // TO FIX: uncomment the next line
   // virtual ~base() {}
};

struct derived : base {
   std::vector<int>::iterator m_iter;
   derived( std::vector<int>::iterator iter ) : m_iter( iter ) {}
   ~derived() {}
};

int main() {
   std::vector<int> vect( 10 );
   base * pb = new derived( vect.begin() );
   delete pb;  // doesn't call ~derived()
   // access violation
}
```

See also

C++ Standard Library Overview

_HAS_ITERATOR_DEBUGGING

Article • 08/03/2021

Superseded by _ITERATOR_DEBUG_LEVEL, this macro defines whether the iterator debugging feature is enabled in a debug build. By default, iterator debugging is enabled in Debug builds and disabled in Retail builds. For more information, see Debug Iterator Support.

> **ⓘ Important**
>
> Direct use of the _HAS_ITERATOR_DEBUGGING macro is deprecated. Instead, use _ITERATOR_DEBUG_LEVEL to control iterator debug settings. For more information, see **_ITERATOR_DEBUG_LEVEL**.

Remarks

To enable iterator debugging in debug builds, set _ITERATOR_DEBUG_LEVEL to 2. This is equivalent to a _HAS_ITERATOR_DEBUGGING setting of 1, or enabled:

```
C++

#define _ITERATOR_DEBUG_LEVEL 2
```

_ITERATOR_DEBUG_LEVEL cannot be set to 2 (and _HAS_ITERATOR_DEBUGGING cannot be set to 1) in retail builds.

To disable debug iterators in debug builds, set _ITERATOR_DEBUG_LEVEL to 0 or 1. This is equivalent to a _HAS_ITERATOR_DEBUGGING setting of 0, or disabled:

```
C++

#define _ITERATOR_DEBUG_LEVEL 0
```

See also

_ITERATOR_DEBUG_LEVEL
Debug Iterator Support
Checked Iterators
Safe Libraries: C++ Standard Library

Thread Safety in the C++ Standard Library

Article • 08/03/2021

The following thread safety rules apply to all classes in the C++ Standard Library—this includes `shared_ptr`, as described below. Stronger guarantees are sometimes provided—for example, the standard iostream objects, as described below, and types intended for multithreading, like those in <atomic>.

An object is thread-safe for reading from multiple threads. For example, given an object A, it is safe to read A from thread 1 and from thread 2 simultaneously.

If an object is being written to by one thread, then all reads and writes to that object on the same or other threads must be protected. For example, given an object A, if thread 1 is writing to A, then thread 2 must be prevented from reading from or writing to A.

It's safe to read and write to one instance of a type even if another thread is reading or writing to a different instance of the same type. For example, given objects A and B of the same type, it's safe when A is being written in thread 1 and B is being read in thread 2.

shared_ptr

Multiple threads can simultaneously read and write different shared_ptr objects, even when the objects are copies that share ownership.

iostream

The standard iostream objects `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr`, and `wclog` follow the same rules as the other classes, with this exception: it's safe to write to an object from multiple threads. For example, thread 1 can write to cout at the same time as thread 2. However, this can cause the output from the two threads to be intermixed.

> **Note**
>
> Reading from a stream buffer is not considered to be a read operation. Instead it's considered to be a write operation because the state of the class is changed.

See also

C++ Standard Library Overview

stdext Namespace

Article • 08/03/2021

Members of the <hash_map> and <hash_set> header files are not currently part of the ISO C++ standard. Therefore, these types and members have been moved from the `std` namespace to namespace `stdext`, to remain conformant with the C++ standard.

When compiling with /Ze, which is the default, the compiler warns on the use of `std` for members of the <hash_map> and <hash_set> header files. To disable the warning, use the warning pragma.

To have the compiler generate an error for the use of `std` for members of the <hash_map> and <hash_set> header files with **/Ze**, add the following directive before you `#include` any C++ Standard Library header files.

```
C++

#define _DEFINE_DEPRECATED_HASH_CLASSES 0
```

When compiling with **/Za**, the compiler generates an error.

See also

C++ Standard Library Overview

C++ Standard Library Containers

Article • 08/03/2021

The Standard Library provides various type-safe containers for storing collections of related objects. The containers are class templates. When you declare a container variable, you specify the type of the elements that the container will hold. Containers can be constructed with initializer lists. They have member functions for adding and removing elements and doing other operations.

You iterate over the elements in a container, and access the individual elements by using iterators. You can use iterators explicitly by using their member functions and operators and global functions. You can also use them implicitly, for example by using a range-for loop. Iterators for all C++ Standard Library containers have a common interface but each container defines its own specialized iterators.

Containers can be divided into three categories: sequence containers, associative containers, and container adapters.

Sequence Containers

Sequence containers maintain the ordering of inserted elements that you specify.

A `vector` container behaves like an array, but can automatically grow as required. It is random access and contiguously stored, and length is highly flexible. For these reasons and more, `vector` is the preferred sequence container for most applications. When in doubt as to what kind of sequence container to use, start by using a vector! For more information, see vector Class.

An `array` container has some of the strengths of `vector`, but the length isn't as flexible. For more information, see array Class.

A `deque` (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. It shares the random-access and flexible-length advantages of `vector`, but isn't contiguous. For more information, see deque Class.

A `list` container is a doubly linked list that enables bidirectional access, fast insertions, and fast deletions anywhere in the container, but you can't randomly access an element in the container. For more information, see list Class.

A `forward_list` container is a singly linked list—the forward-access version of `list`. For more information, see forward_list Class.

Associative Containers

In associative containers, elements are inserted in a pre-defined order—for example, as sorted ascending. Unordered associative containers are also available. The associative containers can be grouped into two subsets: maps and sets.

A `map`, sometimes referred to as a dictionary, consists of a key/value pair. The key is used to order the sequence, and the value is associated with that key. For example, a `map` might contain keys that represent every unique word in a text and corresponding values that represent the number of times that each word appears in the text. The unordered version of `map` is `unordered_map`. For more information, see map Class and unordered_map Class.

A `set` is just an ascending container of unique elements—the value is also the key. The unordered version of `set` is `unordered_set`. For more information, see set Class and unordered_set Class.

Both `map` and `set` only allow one instance of a key or element to be inserted into the container. If multiple instances of elements are required, use `multimap` or `multiset`. The unordered versions are `unordered_multimap` and `unordered_multiset`. For more information, see multimap Class, unordered_multimap Class, multiset Class, and unordered_multiset Class.

Ordered maps and sets support bi-directional iterators, and their unordered counterparts support forward iterators. For more information, see Iterators.

Heterogeneous Lookup in Associative Containers (C++14)

The ordered associative containers (map, multimap, set, and multiset) now support heterogeneous lookup, which means that you're no longer required to pass the exact same object type as the key or element in member functions such as `find()` and `lower_bound()`. Instead, you can pass any type for which an overloaded `operator<` is defined that enables comparison to the key type.

Heterogenous lookup is enabled on an opt-in basis when you specify the `std::less<>` or `std::greater<>` "diamond functor" comparator when declaring the container variable, as shown here:

```
C++

std::set<BigObject, std::less<>> myNewSet;
```

If you use the default comparator, then the container behaves exactly as it did in C++11 and earlier.

The following example shows how to overload `operator<` to enable users of a `std::set` to do lookups simply by passing in a small string that can be compared to each object's `BigObject::id` member.

C++

```
#include <set>
#include <string>
#include <iostream>
#include <functional>

using namespace std;

class BigObject
{
public:
    string id;
    explicit BigObject(const string& s) : id(s) {}
    bool operator< (const BigObject& other) const
    {
        return this->id < other.id;
    }

    // Other members....
};

inline bool operator<(const string& otherId, const BigObject& obj)
{
    return otherId < obj.id;
}

inline bool operator<(const BigObject& obj, const string& otherId)
{
    return obj.id < otherId;
}

int main()
{
    // Use C++14 brace-init syntax to invoke BigObject(string).
    // The s suffix invokes string ctor. It is a C++14 user-defined
    // literal defined in <string>
    BigObject b1{ "42F"s };
    BigObject b2{ "52F"s };
    BigObject b3{ "62F"s };
    set<BigObject, less<>> myNewSet; // C++14
    myNewSet.insert(b1);
    myNewSet.insert(b2);
    myNewSet.insert(b3);
    auto it = myNewSet.find(string("62F"));
    if (it != myNewSet.end())
```

```
        cout << "myNewSet element = " << it->id << endl;
    else
        cout << "element not found " << endl;

    // Keep console open in debug mode:
    cout << endl << "Press Enter to exit.";
    string s;
    getline(cin, s);
    return 0;
}

//Output: myNewSet element = 62F
```

The following member functions in map, multimap, set, and multiset have been overloaded to support heterogeneous lookup:

1. find
2. count
3. lower_bound
4. upper_bound
5. equal_range

Container Adapters

A container adapter is a variation of a sequence or associative container that restricts the interface for simplicity and clarity. Container adapters don't support iterators.

A `queue` container follows FIFO (first in, first out) semantics. The first element *pushed*—that is, inserted into the queue—is the first to be *popped*—that is, removed from the queue. For more information, see queue Class.

A `priority_queue` container is organized such that the element that has the highest value is always first in the queue. For more information, see priority_queue Class.

A `stack` container follows LIFO (last in, first out) semantics. The last element pushed on the stack is the first element popped. For more information, see stack Class.

Because container adapters don't support iterators, they can't be used with the C++ Standard Library algorithms. For more information, see Algorithms.

Requirements for Container Elements

In general, elements inserted into a C++ Standard Library container can be of just about any object type if they are copyable. Movable-only elements—for example, those such as `vector<unique_ptr<T>>` that are created by using `unique_ptr<>` will work as long as you don't call member functions that attempt to copy them.

The destructor isn't permitted to throw an exception.

Ordered associative containers—described earlier in this article—must have a public comparison operator defined. By default, the operator is `operator<`, but even types that don't work with `operator<` are supported.

Some operations on containers might also require a public default constructor and a public equivalence operator. For example, the unordered associative containers require support for equality and hashing.

Accessing Container Elements

The elements of containers are accessed by using iterators. For more information, see Iterators.

> ⓘ **Note**
>
> You can also use **range-based for loops** to iterate over C++ Standard Library collections.

Comparing containers

All containers overload the operator== for comparing two containers of the same type that have the same element type. You can use == to compare a vector<string> to another vector<string>, but you can't use it to compare a vector<string> to a list<string> or a vector<string> to a vector<char*>. In C++98/03, you can use std::equal or std::mismatch to compare dissimilar container types and/or element types. In C++11, you can also use std::is_permutation. But in all these cases the functions assume the containers are the same length. If the second range is shorter than the first, then undefined behavior results. If the second range is longer, results can still be incorrect because the comparison never continues past the end of the first range.

Comparing dissimilar containers (C++14)

In C++14 and later, you can compare dissimilar containers and/or dissimilar elements types by using one of the `std::equal`, `std::mismatch`, or `std::is_permutation` function overloads that take two complete ranges. These overloads enable you to compare containers with different lengths. These overloads are much less susceptible to user error, and are optimized to return false in constant time when containers of dissimilar lengths are compared. Therefore, we recommend you use these overloads unless you have a clear reason not to, or you're using a std::list container, which does not benefit from the dual-range optimizations.

See also

Parallel Containers
<sample container>
Thread Safety in the C++ Standard Library

<sample container>

Article • 12/07/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Shows you the structure of the container headers in the C++ Standard Library.

Syntax

```
C++

#include <sample container> // Nonfunctional header
```

`<sample container>` Operators

Article • 12/07/2021

For more information about the operators in <sample container>, see <sample container>.

operator!=

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads `operator!=` to compare two objects of class template Container.

Syntax

C++

```
template <class Ty>
bool operator!=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(left == right)`.

See also

<sample container>

`operator==` (`<sample container>`)

Article • 12/06/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads `operator==` to compare two objects of class template Container.

Syntax

```
C++

template <class Ty>
bool operator==(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `left.`size `== right.size && equal(left.`begin`, left.`end`, right.begin)`.

See also

<sample container>

`operator< (<sample container>)`

Article • 12/06/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads **operator<** to compare two objects of class template Container.

Syntax

```
C++

template <class Ty>
bool operator<(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `lexicographical_compare(left.begin, left.end, right.begin, right.end)`.

See also

<sample container>
begin
end

`operator<=` (`<sample container>`)

Article • 12/06/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads **operator**<= to compare two objects of class template Container.

Syntax

```
C++

template <class Ty>
bool operator<=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(right < left)`.

See also

<sample container>

`operator>` (`<sample container>`)

Article • 03/10/2022

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads **operator>** to compare two objects of class template Container.

Syntax

```
C++
```

```
template <class Ty>
bool operator*gt(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `right < left`.

See also

<sample container>

`operator>=`

Article • 12/06/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Overloads **operator>=** to compare two objects of class template Container.

Syntax

```
C++
```

```
template <class Ty>
bool operator>=(
    const Container <Ty>& left,
    const Container <Ty>& right);
```

Return Value

Returns `!(left < right)`.

See also

<sample container>

`<sample container>` Specialized Template Functions

Article • 12/07/2021

For more information about the specialized template functions in <sample container>, see <sample container>.

`swap (<sample container>)`

Article • 12/06/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Executes `left.swap(right)`.

Syntax

C++

```
template <class Ty>
void swap(
    Container <Ty>& left,
    Container <Ty>& right);
```

See also

<sample container>

`<sample container>` Classes

Article • 12/07/2021

For more information about the classes in <sample container>, see <sample container>.

Sample Container Class

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that controls a varying-length sequence of elements, typically of type `Ty`. The sequence is stored in different ways, depending on the actual container.

A container constructor or member function may find occasion to call the constructor **Ty(const Ty&)** or the function **Ty::operator=(const Ty&)**. If such a call throws an exception, the container object is obliged to maintain its integrity, and to rethrow any exception it catches. You can safely swap, assign to, erase, or destroy a container object after it throws one of these exceptions. In general, however, you cannot otherwise predict the state of the sequence controlled by the container object.

A few additional caveats:

- If the expression `~Ty` throws an exception, the resulting state of the container object is undefined.
- If the container stores an allocator object *al*, and *al* throws an exception other than as a result of a call to `al.allocate`, the resulting state of the container object is undefined.
- If the container stores a function object *comp*, to determine how to order the controlled sequence, and *comp* throws an exception of any kind, the resulting state of the container object is undefined.

The container classes defined by C++ Standard Library satisfy several additional requirements, as described in the following paragraphs.

Container class template list provides deterministic, and useful, behavior even in the presence of the exceptions described above. For example, if an exception is thrown during the insertion of one or more elements, the container is left unaltered and the exception is rethrown.

For *all* the container classes defined by C++ Standard Library, if an exception is thrown during calls to the following member functions, `insert`, `push_back`, or `push_front`, the

container is left unaltered and the exception is rethrown.

For *all* the container classes defined by C++ Standard Library, no exception is thrown during calls to the following member functions: `pop_back`, `pop_front`.

The member function erase throws an exception only if a copy operation (assignment or copy construction) throws an exception.

Moreover, no exception is thrown while copying an iterator returned by a member function.

The member function swap makes additional promises for *all* container classes defined by C++ Standard Library:

- The member function throws an exception only if the container stores an allocator object al, and `al` throws an exception when copied.
- References, pointers, and iterators that designate elements of the controlled sequences being swapped remain valid.

An object of a container class defined by C++ Standard Library allocates and frees storage for the sequence it controls through a stored object of type `Alloc`, which is typically a template parameter. Such an allocator object must have the same external interface as an object of class `allocator<Ty>`. In particular, `Alloc` must be the same type as `Alloc::rebind<value_type>::other`

For *all* container classes defined by C++ Standard Library, the member function `Alloc get_allocator const;` returns a copy of the stored allocator object. Note that the stored allocator object is *not* copied when the container object is assigned. All constructors initialize the value stored in `allocator`, to `Alloc` if the constructor contains no allocator parameter.

According to the C++ Standard, a container class defined by the C++ Standard Library can assume that:

- All objects of class `Alloc` compare equal.
- Type `Alloc::const_pointer` is the same as `const Ty *`.
- Type `Alloc::const_reference` is the same as `const Ty&`.
- Type `Alloc::pointer` is the same as `Ty *`.
- Type `Alloc::reference` is the same as `Ty&`.

In this implementation, however, containers do not make such simplifying assumptions. Thus, they work properly with allocator objects that are more ambitious:

- All objects of class `Alloc` does not need to compare equal. (You can maintain multiple pools of storage.)
- Type `Alloc::const_pointer` does not need to be the same as `const Ty *`. (A const pointer can be a class.)
- Type `Alloc::pointer` does not need to be the same as `Ty *`. (A pointer can be a class.)

Requirements

Header: <sample container>

See also

<sample container>

Sample Container Members

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Reference

Typedefs

Name	Description
const_iterator	Describes an object that can serve as a constant iterator for the controlled sequence.
const_reference	Describes an object that can serve as a constant reference to an element of the controlled sequence.
const_reverse_iterator	Describes an object that can serve as a constant reverse iterator for the controlled sequence.
difference_type	Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.
iterator	Describes an object that can serve as an iterator for the controlled sequence.
reference	Describes an object that can serve as a reference to an element of the controlled sequence.
reverse_iterator	Describes an object that can serve as a reverse iterator for the controlled sequence.
size_type	Describes an object that can represent the length of any controlled sequence.
value_type	Acts a synonym for the template parameter `Ty`.

Member Functions

Name	Description
begin	Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).
clear	Calls erase(begin, end).
empty	Returns `true` for an empty controlled sequence.
end	Returns an iterator that points just beyond the end of the sequence.
erase	Erases an element.
max_size	Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.
rbegin	Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.
rend	The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.
size	Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.
swap	

Sample Container Typedefs

Article • 08/03/2021

For more information about the typedefs in the sample container class, see Sample Container Class

Container Class::const_iterator

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as a constant iterator for the controlled sequence.

Syntax

```
C++

typedef T6 const_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T6`.

See also

Sample Container Class

Container Class::const_reference

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as a constant reference to an element of the controlled sequence.

Syntax

```
C++

typedef T3 const_reference;
```

Remarks

It is described here as a synonym for the unspecified type `T3` (typically `Alloc::const_reference`).

See also

Sample Container Class

Container Class::const_reverse_iterator

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as a constant reverse iterator for the controlled sequence.

Syntax

```
C++

typedef T8 const_reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T8` (typically reverse_iterator <const_iterator>).

See also

Sample Container Class

Container Class::difference_type

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

Syntax

```
C++

typedef T1 difference_type;
```

Remarks

It is described here as a synonym for the unspecified type `T1` (typically `Alloc::difference_type`).

See also

Sample Container Class

Container Class::iterator

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as an iterator for the controlled sequence.

Syntax

```
C++

typedef T5 iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T5`. An object of type `iterator` can be cast to an object of type const_iterator.

See also

Sample Container Class

Container Class::reference

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as a reference to an element of the controlled sequence.

Syntax

```
C++

typedef T2 reference;
```

Remarks

It is described here as a synonym for the unspecified type `T2` (typically `Alloc::reference`). An object of type `reference` can be cast to an object of type const_reference.

See also

Sample Container Class

Container Class::reverse_iterator

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can serve as a reverse iterator for the controlled sequence.

Syntax

```
C++

typedef T7 reverse_iterator;
```

Remarks

It is described here as a synonym for the unspecified type `T7` (typically `reverse_iterator` <iterator>).

See also

Sample Container Class

Container Class::size_type

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Describes an object that can represent the length of any controlled sequence.

Syntax

```
C++

typedef T0 size_type;
```

Remarks

It is described here as a synonym for the unspecified type `T0` (typically `Alloc::size_type`).

See also

Sample Container Class

Container Class::value_type

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Acts a synonym for the template parameter *Ty*.

Syntax

```
C++

typedef T4 value_type;
```

Remarks

It is described here as a synonym for the unspecified type `T4` (typically `Alloc::value_type`).

See also

Sample Container Class

Sample Container Member Functions

Article • 08/03/2021

For more information about the member functions in the sample container class, see Sample Container Class

Container Class::begin

Article • 08/03/2021

> **Note**
>
> This topic is in the Visual Studio C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Syntax

```
C++

const_iterator begin() const;

iterator begin();
```

See also

Sample Container Class

Container Class::clear

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Calls erase(begin, end).

Syntax

```
C++

void clear();
```

See also

Sample Container Class

Container Class::empty

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns `true` for an empty controlled sequence.

Syntax

C++

```
bool empty() const;
```

See also

Sample Container Class

Container Class::end

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns an iterator that points just beyond the end of the sequence.

Syntax

```
C++

const_iterator end() const;

iterator end();
```

See also

Sample Container Class

Container Class::erase

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Erases an element.

Syntax

```
C++
```

```
iterator erase(
    iterator _Where);

iterator erase(
    iterator first,
    iterator last);
```

Remarks

The first member function removes the element of the controlled sequence pointed to by *_Where*. The second member function removes the elements of the controlled sequence in the range [`first`, `last`). Both return an iterator that designates the first element remaining beyond any elements removed, or end if no such element exists.

The member functions throw an exception only if a copy operation throws an exception.

See also

Sample Container Class

Container Class::max_size

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns the length of the longest sequence that the object can control, in constant time regardless of the length of the controlled sequence.

Syntax

```
C++

size_type max_size() const;
```

See also

Sample Container Class

Container Class::rbegin

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns a reverse iterator that points just beyond the end of the controlled sequence, designating the beginning of the reverse sequence.

Syntax

```
C++

const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

See also

Sample Container Class

Container Class::rend

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence), designating the end of the reverse sequence.

Syntax

```
C++

const_reverse_iterator rend() const;

reverse_iterator rend();
```

See also

Sample Container Class

Container Class::size

Article • 08/03/2021

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Returns the length of the controlled sequence, in constant time regardless of the length of the controlled sequence.

Syntax

```
C++

size_type size() const;
```

See also

Sample Container Class

Container Class::swap

Article • 06/16/2022

> **Note**
>
> This topic is in the Microsoft C++ documentation as a nonfunctional example of containers used in the C++ Standard Library. For more information, see **C++ Standard Library Containers**.

Swaps the controlled sequences between `*this` and its argument.

Syntax

C++

```
void swap(Container& right);
```

Remarks

If `*this.get_allocator == right.get_allocator`, it does a swap in constant time. Otherwise, it performs element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

See also

Sample Container Class

Iterators

Article • 01/28/2022

An iterator is an object that can iterate over elements in a C++ Standard Library container and provide access to individual elements. The C++ Standard Library containers all provide iterators so that algorithms can access their elements in a standard way without having to be concerned with the type of container the elements are stored in.

You can use iterators explicitly using member and global functions such as `begin()` and `end()` and operators such as `++` and `--` to move forward or backward. You can also use iterators implicitly with a range-for loop or (for some iterator types) the subscript operator `[]`.

In the C++ Standard Library, the beginning of a sequence or range is the first element. The end of a sequence or range is always defined as one past the last element. The global functions `begin` and `end` return iterators to a specified container. The typical explicit iterator loop over all elements in a container looks like this:

C++

```
vector<int> vec{ 0,1,2,3,4 };
for (auto it = begin(vec); it != end(vec); it++)
{
    // Access element using dereference operator
    cout << *it << " ";
}
```

The same thing can be accomplished more simply with a range-for loop:

C++

```
for (auto num : vec)
{
    // no dereference operator
    cout << num << " ";
}
```

There are five categories of iterators. In order of increasing power, the categories are:

- **Output**. An *output iterator* `X` can iterate forward over a sequence by using the `++` operator, and can write an element only once, by using the `*` operator.

- **Input**. An *input iterator* `X` can iterate forward over a sequence by using the `++` operator, and can read an element any number of times by using the `*` operator. You can compare input iterators by using the `==` and `!=` operators. After you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented afterwards.

- **Forward**. A *forward iterator* `X` can iterate forward over a sequence using the ++ operator and can read any element or write non-const elements any number of times by using the `*` operator. You can access element members by using the `->` operator and compare forward iterators by using the `==` and `!=` operators. You can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently. A forward iterator that is initialized without reference to any container is called a *null forward iterator*. Null forward iterators always compare equal.

- **Bidirectional**. A *bidirectional iterator* `X` can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in `--X`, `X--`, or `(V = *X--)`. You can access element members and compare bidirectional iterators in the same way as forward iterators.

- **Random access**. A *random-access iterator* `X` can take the place of a bidirectional iterator. With a random access iterator, you can use the subscript operator `[]` to access elements. You can use the `+`, `-`, `+=` and `-=` operators to move forward or backward a specified number of elements and to calculate the distance between iterators. You can compare bidirectional iterators by using `==`, `!=`, `<`, `>`, `<=`, and `>=`.

All iterators can be assigned or copied. They're assumed to be lightweight objects and are often passed and returned by value, not by reference. Note also that none of the operations previously described can throw an exception when performed on a valid iterator.

The hierarchy of iterator categories can be summarized by showing three sequences. For write-only access to a sequence, you can use any of:

> output iterator
> -> forward iterator
> -> bidirectional iterator
> -> random-access iterator

The right arrow means "can be replaced by." Any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

> input iterator
> -> forward iterator
> -> bidirectional iterator
> -> random-access iterator

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

> forward iterator
> -> bidirectional iterator
> -> random-access iterator

An object pointer can always serve as a random-access iterator, so it can serve as any category of iterator if it supports the proper read/write access to the sequence it designates.

An iterator `Iterator` other than an object pointer must also define the member types required by the specialization `iterator_traits<Iterator>`. These requirements can be met by deriving `Iterator` from the public base class iterator.

It's important to understand the promises and limitations of each iterator category to see how iterators are used by containers and algorithms in the C++ Standard Library.

> ⓘ **Note**
>
> You can avoid using iterators explicitly by using range-for loops. For more information, see **Range-based for statement**.

Microsoft C++ now offers checked iterators and debug iterators to ensure that you do not overwrite the bounds of your container. For more information, see Checked Iterators and Debug Iterator Support.

See also

Algorithms

Article • 01/30/2023

Algorithms are a fundamental part of the C++ Standard Library. Algorithms don't work with containers themselves but rather with iterators. Therefore, the same algorithm can be used by most if not all of the C++ Standard Library containers. This section discusses the conventions and terminology of the C++ Standard Library algorithms.

Remarks

The descriptions of the algorithm function templates employ several shorthand phrases:

- The phrase "in the range [*A*, *B*)" means the sequence of zero or more discrete values beginning with *A* up to but not including *B*. A range is valid only if *B* is reachable from *A*, you can store *A* in an object *N* (*N* = *A*), you can increment the object zero or more times (++*N*), and have the object compare equal to *B* after a finite number of increments (*N* == *B*).
- The phrase "each *N* in the range [*A*, *B*)" means that *N* begins with the value *A* and is incremented zero or more times until it equals the value *B*. The case *N* == *B* isn't in the range.
- The phrase "the lowest value of *N* in the range [*A*, *B*) such that *X*" means that the condition *X* is determined for each *N* in the range [*A*, *B*) until the condition *X* is met.
- The phrase "the highest value of *N* in the range [*A*, *B*) such that *X*" means that *X* is determined for each *N* in the range [*A*, *B*). The function stores in *K* a copy of *N* each time the condition *X* is met. If any such store occurs, the function replaces the final value of *N*, which equals *B*, with the value of *K*. For a bidirectional or random-access iterator, however, it can also mean that *N* begins with the highest value in the range and is decremented over the range until the condition *X* is met.
- Expressions such as *X* - *Y*, where *X* and *Y* can be iterators other than random-access iterators, are intended in the mathematical sense. The function doesn't necessarily evaluate operator - if it must determine such a value. The same is also true for expressions such as *X* + *N* and *X* - *N*, where *N* is an integer type.

Several algorithms make use of a predicate that performs a pairwise comparison, such as with `operator==`, to yield a **`bool`** result. The predicate function `operator==`, or any replacement for it, must not alter either of its operands. It must yield the same **`bool`**

result every time it's evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

Several algorithms make use of a predicate that must impose a strict weak ordering on pairs of elements from a sequence. For the predicate *pred*(*X*, *Y*):

- Strict means that *pred*(*X*, *X*) is false.
- Weak means that *X* and *Y* have an equivalent ordering if !*pred*(*X*, *Y*) && !*pred*(*Y*, *X*) (*X* == *Y* doesn't need to be defined).
- Ordering means that *pred*(*X*, *Y*) && *pred*(*Y*, *Z*) implies *pred*(*X*, *Z*).

Some of these algorithms implicitly use the predicate *X* < *Y*. Other predicates that typically satisfy the strict weak ordering requirement are *X* > *Y*, `less`(*X*, *Y*), and `greater`(*X*, *Y*). Note, however, that predicates such as *X* <= *Y* and *X* >= *Y* don't satisfy this requirement.

A sequence of elements designated by iterators in the range [*First*, *Last*) is a sequence ordered by operator `<` if, for each *N* in the range [0, *Last* - *First*) and for each *M* in the range (*N*, *Last* - *First*) the predicate !(*(*First* + *M*) < *(*First* + *N*)) is true. (Note that the elements are sorted in ascending order.) The predicate function `operator<`, or any replacement for it, must not alter either of its operands. It must yield the same **`bool`** result every time it's evaluated, and it must yield the same result if a copy of either operand is substituted for the operand. Moreover, it must impose a strict weak ordering on the operands it compares.

A sequence of elements designated by iterators in the range [`First`, `Last`) is a heap ordered by `operator<` if, for each *N* in the range [1, *Last* - *First*) the predicate !(**First* < *(*First* + *N*)) is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions make_heap, pop_heap, and push_heap. As with an ordered sequence, the predicate function `operator<`, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering on the operands it compares. It must yield the same **`bool`** result every time it's evaluated, and it must yield the same result if a copy of either operand is substituted for the operand.

The C++ Standard Library algorithms are located in the <algorithm> and <numeric> header files.

See also

Allocators

Article • 02/11/2022

Allocators are used by the C++ Standard Library to handle the allocation and deallocation of elements stored in containers. All C++ Standard Library containers except `std::array` have a template parameter of type `allocator<Type>`, where `Type` represents the type of the container element. For example, the `vector` class is declared as follows:

C++

```
template <
    class Type,
    class Allocator = allocator<Type>
>
class vector
```

The C++ Standard Library provides a default implementation for an allocator. In C++11 and later, the default allocator is updated to expose a smaller interface; the new allocator is called a *minimal allocator*. In particular, the minimal allocator's `construct()` member supports move semantics, which can greatly improve performance. In most cases, this default allocator should be sufficient. In C++11 all the Standard Library types and functions that take an allocator type parameter support the minimal allocator interface, including `std::function`, `shared_ptr, allocate_shared()`, and `basic_string`. For more information on the default allocator, see allocator Class.

Writing Your Own Allocator (C++11)

The default allocator uses `new` and `delete` to allocate and deallocate memory. If you want to use a different method of memory allocation, such as using shared memory, then you must create your own allocator. If you are targeting C++11 and you need to write a new custom allocator, make it a minimal allocator if possible. Even if you have already implemented an old-style allocator, consider modifying it to be a *minimal allocator* in order to take advantage of the more efficient `construct()` method that will be provided for you automatically.

A minimal allocator requires much less boilerplate and enables you to focus on the `allocate` and `deallocate` member functions, which do all of the work. When creating a minimal allocator, do not implement any members except the ones shown in the example below:

1. a converting copy constructor (see example)
2. `operator==`
3. `operator!=`
4. `allocate`
5. `deallocate`

The C++11 default `construct()` member that will be provided for you does perfect forwarding and enables move semantics; it is much more efficient in many cases than the older version.

> ⚠ **Warning**
>
> At compile time, the C++ Standard Library uses the `allocator_traits` class to detect which members you have explicitly provided and provides a default implementation for any members that are not present. Do not interfere with this mechanism by providing a specialization of `allocator_traits` for your allocator!

The following example shows a minimal implementation of an allocator that uses `malloc` and `free`. Note the use of the new exception type `std::bad_array_new_length` which is thrown if the array size is less than zero or greater than the maximum allowed size.

C++

```
#pragma once
#include <stdlib.h> //size_t, malloc, free
#include <new> // bad_alloc, bad_array_new_length
#include <memory>
template <class T>
struct Mallocator
{
    typedef T value_type;
    Mallocator() noexcept {} //default ctor not required by C++ Standard
Library

    // A converting copy constructor:
    template<class U> Mallocator(const Mallocator<U>&) noexcept {}
    template<class U> bool operator==(const Mallocator<U>&) const noexcept
    {
        return true;
    }
    template<class U> bool operator!=(const Mallocator<U>&) const noexcept
    {
```

```
        return false;
    }
    T* allocate(const size_t n) const;
    void deallocate(T* const p, size_t) const noexcept;
};

template <class T>
T* Mallocator<T>::allocate(const size_t n) const
{
    if (n == 0)
    {
        return nullptr;
    }
    if (n > static_cast<size_t>(-1) / sizeof(T))
    {
        throw std::bad_array_new_length();
    }
    void* const pv = malloc(n * sizeof(T));
    if (!pv) { throw std::bad_alloc(); }
    return static_cast<T*>(pv);
}

template<class T>
void Mallocator<T>::deallocate(T * const p, size_t) const noexcept
{
    free(p);
}
```

Writing Your Own Allocator (C++03)

In C++03, any allocator used with C++ Standard Library containers must implement the following type definitions:

`const_pointer`

`const_reference`

`difference_type`

`pointer`

`rebind`

`reference`

`size_type`

`value_type`

In addition, any allocator used with C++ Standard Library containers must implement the following methods:

Constructor
Copy constructor
Destructor

`address`
`allocate`
`construct`

`deallocate`
`destroy`
`max_size`

`operator!=`
`operator==`

For more information on these type definitions and methods, see allocator Class.

See also

C++ Standard Library Reference

Function Objects in the C++ Standard Library

Article • 08/03/2021

A *function object*, or *functor*, is any type that implements operator(). This operator is referred to as the *call operator* or sometimes the *application operator*. The C++ Standard Library uses function objects primarily as sorting criteria for containers and in algorithms.

Function objects provide two main advantages over a straight function call. The first is that a function object can contain state. The second is that a function object is a type and therefore can be used as a template parameter.

Creating a Function Object

To create a function object, create a type and implement operator(), such as:

C++

```
class Functor
{
public:
    int operator()(int a, int b)
    {
        return a < b;
    }
};

int main()
{
    Functor f;
    int a = 5;
    int b = 7;
    int ans = f(a, b);
}
```

The last line of the `main` function shows how you call the function object. This call looks like a call to a function, but it's actually calling operator() of the Functor type. This similarity between calling a function object and a function is how the term function object came about.

Function Objects and Containers

The C++ Standard Library contains several function objects in the <functional> header file. One use of these function objects is as a sorting criterion for containers. For example, the `set` container is declared as follows:

```
C++

template <class Key,
    class Traits=less<Key>,
    class Allocator=allocator<Key>>
class set
```

The second template argument is the function object `less`. This function object returns `true` if the first parameter is less than the second parameter. Since some containers sort their elements, the container needs a way of comparing two elements. The comparison is done by using the function object. You can define your own sorting criteria for containers by creating a function object and specifying it in the template list for the container.

Function Objects and Algorithms

Another use of functional objects is in algorithms. For example, the `remove_if` algorithm is declared as follows:

```
C++

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(
    ForwardIterator first,
    ForwardIterator last,
    Predicate pred);
```

The last argument to `remove_if` is a function object that returns a boolean value (a *predicate*). If the result of the function object is `true`, then the element is removed from the container being accessed by the iterators `first` and `last`. You can use any of the function objects declared in the <functional> header for the argument `pred` or you can create your own.

See also

C++ Standard Library Reference

`iostream` Programming

Article • 08/03/2021

This section provides a general description of the iostream classes and then describes output streams, input streams, and input/output streams. The end of the section provides information about advanced iostream programming.

There's also a discussion on Thread Safety in the C++ Standard Library and the stdext namespace.

In This Section

What a Stream Is

Output Streams

Input Streams

Input/Output Streams

Custom Manipulators with Arguments

See also

C++ Standard Library
iostreams Conventions

String and I/O Formatting (Modern C++)

Article • 08/03/2021

C++ <iostream> classes, functions, and operators support formatted string I/O. For example, the following code shows how to set `cout` to format an integer to output in hexadecimal. First, it saves the current state to reset it afterwards, because once format state is passed to `cout`, it stays that way until changed. It doesn't just apply to the one line of code.

C++

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    ios state(nullptr);

    cout << "The answer in decimal is: " << 42 << endl;

    state.copyfmt(cout); // save current formatting
    cout << "In hex: 0x" // now load up a bunch of formatting modifiers
        << hex
        << uppercase
        << setw(8)
        << setfill('0')
        << 42            // the actual value we wanted to print out
        << endl;
    cout.copyfmt(state); // restore previous formatting
}
```

This approach is type-safe and extensible, but it's also complex and verbose.

Alternative format options

As an alternative, you can use `Boost.Format` from the Boost C++ libraries, even though it's nonstandard. You can download any Boost library from the Boost website.

Some advantages of `Boost.Format` are:

- Safe: Type-safe, and throws an exception for errors, for example, the specification of too few or too many items.
- Extensible: Works for any type that can be streamed.
- Convenient: Standard POSIX and similar format strings.

Although `Boost.Format` is built on C++ <iostream> facilities, which are safe and extensible, they aren't performance-optimized. When you require performance optimization, consider C printf and sprintf, which are fast and easy to use. However, they aren't extensible or safe from vulnerabilities. (Safe versions exist, but they incur a slight performance penalty. For more information, see printf_s, _printf_s_l, wprintf_s, _wprintf_s_l and sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l).

The following code demonstrates some of the Boost formatting features.

C++

```
    string s = str( format("%2% %2% %1%\n") % "world" % "hello" );
    // s contains "hello hello world"

    for( auto i = 0; i < names.size(); ++i )
        cout << format("%1% %2% %|40t|%3%\n") % first[i] % last[i] % tel[i];
    // Georges Benjamin Clemenceau             +33 (0) 123 456 789
    // Jean de Lattre de Tassigny              +33 (0) 987 654 321
```

See also

Welcome back to C++
C++ Language Reference
C++ Standard Library
<iostream>
<limits>
<iomanip>

What a Stream Is

Article • 03/07/2022

Like C, C++ doesn't have built-in input/output capability. All C++ compilers, however, come bundled with a systematic, object-oriented I/O package, known as the `iostream` classes. The stream is the central concept of the `iostream` classes. You can think of a stream object as a smart file that acts as a source and destination for bytes. A stream's characteristics are determined by its class and by customized insertion and extraction operators.

Through device drivers, the disk operating system deals with the keyboard, screen, printer, and communication ports as extended files. The `iostream` classes interact with these extended files. Built-in classes support reading from and writing to memory with syntax identical to that for disk I/O, which makes it easy to derive stream classes.

In This Section

Input/Output Alternatives

See also

iostream Programming

Input/Output Alternatives

Article • 08/03/2021

The Microsoft C++ compiler provides several alternatives for I/O programming:

- C run-time library direct, unbuffered I/O.
- ANSI C run-time library stream I/O.
- Console and port direct I/O.
- Microsoft Foundation Class Library.
- Microsoft C++ Standard Library.

The iostream classes are useful for buffered, formatted text I/O. They are also useful for unbuffered or binary I/O if you need a C++ programming interface and decide not to use the Microsoft Foundation Class (MFC) library. The iostream classes are an object-oriented I/O alternative to the C run-time functions.

You can use iostream classes with the Microsoft Windows operating system. String and file streams work without restrictions, but the character-mode stream objects `cin`, `cout`, `cerr`, and `clog` are inconsistent with the Windows graphical user interface. You can also derive custom stream classes that interact directly with the Windows environment.

See also

What a Stream Is

Output Streams

Article • 02/11/2022

An output stream object is a destination for bytes. The three most important output stream classes are `ostream`, `ofstream`, and `ostringstream`.

The `ostream` class, through the derived class `basic_ostream`, supports the predefined stream objects:

- `cout` standard output
- `cerr` standard error with limited buffering
- `clog` similar to `cerr` but with full buffering

Objects are rarely constructed from `ostream`; predefined objects are generally used. In some cases, you can reassign predefined objects after program startup. The `ostream` class, which can be configured for buffered or unbuffered operation, is best suited to sequential text-mode output. All functionality of the base class, `ios`, is included in `ostream`. If you construct an object of class `ostream`, you must specify a `streambuf` object to the constructor.

The `ofstream` class supports disk file output. If you need an output-only disk, construct an object of class `ofstream`. You can specify whether `ofstream` objects accept binary or text-mode data when constructing the `ofstream` object or when calling the `open` member function of the object. Many formatting options and member functions apply to `ofstream` objects, and all functionality of the base classes `ios` and `ostream` is included.

If you specify a filename in the constructor, that file is automatically opened when the object is constructed. Otherwise, you can use the `open` member function after invoking the default constructor.

Like the run-time function `sprintf_s`, the `ostringstream` class supports output to in-memory strings. To create a string in memory by using I/O stream formatting, construct an object of class `ostringstream`.

In This Section

Constructing Output Stream Objects

Using Insertion Operators and Controlling Format

Output File Stream Member Functions

Effects of Buffering

Binary Output Files

Overloading the << Operator for Your Own Classes

Writing Your Own Manipulators Without Arguments

See also

ofstream
ostringstream
iostream Programming

Constructing Output Stream Objects

Article • 08/03/2021

If you use only the predefined `cout`, `cerr`, or `clog` objects, you do not need to construct an output stream. You must use constructors for:

- Output File Stream Constructors
- Output String Stream Constructors

Output File Stream Constructors

You can construct an output file stream in one of two ways:

- Use the default constructor, and then call the `open` member function.

```
C++

ofstream myFile; // Static or on the stack
myFile.open("filename");

ofstream* pmyFile = new ofstream; // On the heap
pmyFile->open("filename");
```

- Specify a filename and mode flags in the constructor call.

```
C++

ofstream myFile("filename", ios_base::out);
```

Output String Stream Constructors

To construct an output string stream, you can use `ostringstream` in the following way:

```
C++

using namespace std;
// ...
ostringstream myString;
myString << "this is a test" << ends;

string sp = myString.str(); // Obtain string
cout << sp << endl;
```

The `ends` "manipulator" adds the necessary terminating null character to the string.

See also

Output Streams

Using Insertion Operators and Controlling Format

Article • 10/07/2021

This topic shows how to control format and how to create insertion operators for your own classes. The insertion (`<<`) operator, which is preprogrammed for all standard C++ data types, sends bytes to an output stream object. Insertion operators work with predefined "manipulators," which are elements that change the default format of integer arguments.

You can control the format with the following options:

- Output Width
- Alignment
- Precision
- Radix

Output Width

To align output, you specify the output width for each item by placing the `setw` manipulator in the stream or by calling the `width` member function. This example right-aligns the values in a column at least 10 characters wide:

C++

```
// output_width.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   for( int i = 0; i < 4; i++ )
   {
      cout.width(10);
      cout << values[i] << '\n';
   }
}
```

Output

```
      1.23
     35.36
     653.7
   4358.24
```

Leading blanks are added to any value fewer than 10 characters wide.

To pad a field, use the `fill` member function, which sets the value of the padding character for fields that have a specified width. The default is a blank. To pad the column of numbers with asterisks, modify the previous `for` loop as follows:

C++

```
for (int i = 0; i <4; i++)
{
    cout.width(10);
    cout.fill('*');
    cout << values[i] << endl;
}
```

The `endl` manipulator replaces the newline character (`'\n'`). The output looks like this:

Output

```
******1.23
*****35.36
*****653.7
***4358.24
```

To specify widths for data elements in the same line, use the `setw` manipulator:

C++

```
// setw.cpp
// compile with: /EHsc
#include <iostream>
#include <iomanip>
using namespace std;

int main( )
{
   double values[] = { 1.23, 35.36, 653.7, 4358.24 };
   const char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
   for( int i = 0; i < 4; i++ )
      cout << setw( 7 )  << names[i]
           << setw( 10 ) << values[i] << endl;
}
```

The `width` member function is declared in `<iostream>`. If you use `setw` or any other manipulator with arguments, you must include `<iomanip>`. In the output, strings print in a field of width 6 and integers in a field of width 10:

Output

```
   Zoot      1.23
  Jimmy     35.36
     Al     653.7
   Stan   4358.24
```

`setw` and `width` don't truncate values. If formatted output exceeds the width, the entire value prints, subject to the stream's precision setting. Both `setw` and `width` affect the following field only. Field width reverts to its default behavior (the necessary width) after one field has been printed. However, the other stream format options remain in effect until changed.

Alignment

Output streams default to right-aligned text. To left-align the names in the previous example and right-align the numbers, replace the `for` loop as follows:

C++

```
for (int i = 0; i <4; i++)
    cout << setiosflags(ios::left)
         << setw(6) << names[i]
         << resetiosflags(ios::left)
         << setw(10) << values[i] << endl;
```

The output looks like this:

Output

```
Zoot        1.23
Jimmy      35.36
Al         653.7
Stan     4358.24
```

The left-align flag is set by using the setiosflags manipulator with the `left` enumerator. This enumerator is defined in the ios class, so its reference must include the `ios::` prefix. The resetiosflags manipulator turns off the left-align flag. Unlike `width` and `setw`, the effect of `setiosflags` and `resetiosflags` is permanent.

Precision

The default value for floating-point precision is six. For example, the number 3466.9768 prints as 3466.98. To change the way this value prints, use the setprecision manipulator. The manipulator has two flags: fixed and scientific. If fixed is set, the number prints as 3466.976800. If `scientific` is set, it prints as 3.4669773+003.

To display the floating-point numbers shown in Alignment with one significant digit, replace the `for` loop as follows:

C++

```
for (int i = 0; i <4; i++)
    cout << setiosflags(ios::left)
         << setw(6)
         << names[i]
         << resetiosflags(ios::left)
         << setw(10)
         << setprecision(1)
         << values[i]
         << endl;
```

The program prints this list:

Output

```
Zoot          1
Jimmy      4e+01
Al         7e+02
Stan       4e+03
```

To eliminate scientific notation, insert this statement before the `for` loop:

C++

```
cout << setiosflags(ios::fixed);
```

With fixed notation, the program prints with one digit after the decimal point.

Output

```
Zoot          1.2
Jimmy        35.4
Al          653.7
Stan       4358.2
```

If you change the `ios::fixed` flag to `ios::scientific`, the program prints this:

```
C++

Zoot    1.2e+00
Jimmy   3.5e+01
Al      6.5e+02
Stan    4.4e+03
```

Again, the program prints one digit after the decimal point. If either `ios::fixed` or `ios::scientific` is set, the precision value determines the number of digits after the decimal point. If neither flag is set, the precision value determines the total number of significant digits. The `resetiosflags` manipulator clears these flags.

Radix

The `dec`, `oct`, and `hex` manipulators set the default radix for input and output. For example, if you insert the `hex` manipulator into the output stream, the object correctly translates the internal data representation of integers into a hexadecimal output format. The numbers are displayed with digits a through f in lower case if the uppercase flag is clear (the default); otherwise, they're displayed in upper case. The default radix is `dec` (decimal).

Quoted strings (C++14)

When you insert a string into a stream, you can easily retrieve the same string back by calling the `stringstream::str()` member function. However, if you want to use the extraction operator to insert the stream into a new string at a later point, you may get an unexpected result because the `>>` operator by default will stop when it finds the first whitespace character.

```
C++

std::stringstream ss;
std::string inserted = "This is a sentence.";
std::string extracted;

ss << inserted;
ss >> extracted;

std::cout << inserted;     //  This is a sentence.
std::cout << extracted;    //  This
```

This behavior can be overcome manually, but to make string round-tripping more convenient, C++14 adds the `std::quoted` stream manipulator in `<iomanip>`. Upon insertion, `quoted()` surrounds the string with a delimiter (double quote ' " ' by default) and upon extraction manipulates the stream to extract all characters until the final delimiter is found. Any embedded quotes are escaped with an escape character ('\\' by default).

The delimiters are present only in the stream object; they aren't present in the extracted string but they're present in the string returned by basic_stringstream::str.

The whitespace behavior of the insertion and extraction operations is independent of how a string is represented in code, so the quoted operator is useful regardless of whether the input string is a raw string literal or a regular string. The input string, whatever its format, can have embedded quotes, line breaks, tabs, and so on, and all these will be preserved by the `quoted()` manipulator.

For more information and full code examples, see quoted.

See also

Output Streams

Output File Stream Member Functions

Article • 03/01/2022

Output stream member functions have three types: those that are equivalent to manipulators, those that perform unformatted write operations, and those that otherwise modify the stream state and have no equivalent manipulator or insertion operator. For sequential, formatted output, you might use only insertion operators and manipulators. For random-access binary disk output, you use other member functions, with or without insertion operators.

The open Function for Output Streams

To use an output file stream (ofstream), you must associate that stream with a specific disk file in the constructor or the `open` function. If you use the `open` function, you can reuse the same stream object with a series of files. In either case, the arguments describing the file are the same.

When you open the file associated with an output stream, you generally specify an `open_mode` flag. You can combine these flags, which are defined as enumerators in the `ios` class, with the bitwise OR (`|`) operator. See ios_base::openmode for a list of the enumerators.

Three common output stream situations involve mode options:

- Creating a file. If the file already exists, the old version is deleted.

  C++

  ```
  ofstream ofile("FILENAME");
  // Default is ios::out

  ofstream ofile("FILENAME", ios::out);
  // Equivalent to above
  ```

- Appending records to an existing file or creating one if it doesn't exist.

  C++

  ```
  ofstream ofile("FILENAME", ios::app);
  ```

- Opening two files, one at a time, on the same stream.

```
C++

ofstream ofile();
ofile.open("FILE1", ios::in);
// Do some output
ofile.close();    // FILE1 closed
ofile.open("FILE2", ios::in);
// Do some more output
ofile.close();    // FILE2 closed
// When ofile goes out of scope it is destroyed.
```

The `put` function

The `put` function writes one character to the output stream. The following two statements are the same by default, but the second is affected by the stream's format arguments:

```
C++

cout.put('A');

// Exactly one character written
cout <<'A'; // Format arguments 'width' and 'fill' apply
```

The `write` function

The `write` function writes a block of memory to an output file stream. The length argument specifies the number of bytes written. This example creates an output file stream and writes the binary value of the `Date` structure to it:

```
C++

// write_function.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;

struct Date
{
   int mo, da, yr;
};

int main( )
{
   Date dt = { 6, 10, 92 };
   ofstream tfile( "date.dat" , ios::binary );
```

```
    tfile.write( (char *) &dt, sizeof dt );
}
```

The `write` function doesn't stop when it reaches a `NULL` character, so the complete class structure is written. The function takes two arguments: a `char` pointer and a count of characters to write. Note the required cast to `char *` before the address of the structure object.

The seekp and tellp Functions

An output file stream keeps an internal pointer that points to the position where data is to be written next. The `seekp` member function sets this pointer and thus provides random-access disk file output. The `tellp` member function returns the file position. For examples that use the input stream equivalents to `seekp` and `tellp`, see The seekg and tellg Functions.

The close Function for Output Streams

The `close` member function closes the disk file associated with an output file stream. The file must be closed to complete all disk output. If necessary, the `ofstream` destructor closes the file for you, but you can use the `close` function if you need to open another file for the same stream object.

The output stream destructor automatically closes a stream's file only if the constructor or the `open` member function opened the file. If you pass the constructor a file descriptor for an already-open file or use the `attach` member function, you must close the file explicitly.

Error Processing Functions

Use these member functions to test for errors while writing to a stream:

Function	Return value
bad	Returns `true` if there's an unrecoverable error.
fail	Returns `true` if there's an unrecoverable error or an "expected" condition, such as a conversion error, or if the file isn't found. Processing can often resume after a call to `clear` with a zero argument.

Function	Return value
good	Returns `true` if there's no error condition (unrecoverable or otherwise) and the end-of-file flag isn't set.
eof	Returns `true` on the end-of-file condition.
clear	Sets the internal error state. If called with the default arguments, it clears all error bits.
[`rdstate`](basic-ios-class.md#rdstate	Returns the current error state.

The `!` operator is overloaded to perform the same function as the `fail` function. Thus the expression:

```
C++

if(!cout)...
```

is equivalent to:

```
C++

if(cout.fail())...
```

The `void*()` operator is overloaded to be the opposite of the `!` operator; thus the expression:

```
C++

if(cout)...
```

is equal to:

```
C++

if(!cout.fail())...
```

The `void*()` operator isn't equivalent to `good` because it doesn't test for the end of file.

See also

Output Streams

Effects of Buffering

Article • 08/03/2021

The following example shows the effects of buffering. You might expect the program to print `please wait`, wait 5 seconds, and then proceed. It will not necessarily work this way, however, because the output is buffered.

C++

```
// effects_buffering.cpp
// compile with: /EHsc
#include <iostream>
#include <time.h>
using namespace std;

int main( )
{
   time_t tm = time( NULL ) + 5;
   cout << "Please wait...";
   while ( time( NULL ) < tm )
      ;
   cout << "\nAll done" << endl;
}
```

To make the program work logically, the `cout` object must empty itself when the message is to appear. To flush an `ostream` object, send it the `flush` manipulator:

C++

```
cout <<"Please wait..." <<flush;
```

This step flushes the buffer, ensuring the message prints before the wait. You can also use the `endl` manipulator, which flushes the buffer and outputs a carriage return-line feed, or you can use the `cin` object. This object (with the `cerr` or `clog` objects) is usually tied to the `cout` object. Thus, any use of `cin` (or of the `cerr` or `clog` objects) flushes the `cout` object.

See also

Output Streams

Binary Output Files

Article • 08/03/2021

Streams were originally designed for text, so the default output mode is text. In text mode, the line feed (newline) character expands to a carriage return-line feed pair. The expansion can cause problems, as shown here:

C++

```
// binary_output_files.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };
int main( )
{
    ofstream os( "test.dat" );
    os.write( (char *) iarray, sizeof( iarray ) );
}
```

You might expect this program to output the byte sequence { 99, 0, 10, 0 }; instead, it outputs { 99, 0, 13, 10, 0 }, which causes problems for a program expecting binary input. If you need true binary output, in which characters are written untranslated, you could specify binary output by using the ofstream constructor `openmode` argument:

C++

```
// binary_output_files2.cpp
// compile with: /EHsc
#include <fstream>
using namespace std;
int iarray[2] = { 99, 10 };

int main()
{
   ofstream ofs ( "test.dat", ios_base::binary );

   // Exactly 8 bytes written
   ofs.write( (char*)&iarray[0], sizeof(int)*2 );
}
```

See also

Output Streams

Overloading the `<<` Operator for Your Own Classes

Article • 12/06/2021

Output streams use the insertion (`<<`) operator for standard types. You can also overload the `<<` operator for your own classes.

Example

The `write` function example showed the use of a `Date` structure. A date is an ideal candidate for a C++ class in which the data members (month, day, and year) are hidden from view. An output stream is the logical destination for displaying such a structure. This code displays a date using the `cout` object:

C++

```
Date dt(1, 2, 92);

cout <<dt;
```

To get `cout` to accept a `Date` object after the insertion operator, overload the insertion operator to recognize an `ostream` object on the left and a `Date` on the right. The overloaded `<<` operator function must then be declared as a friend of class `Date` so it can access the private data within a `Date` object.

C++

```
// overload_date.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Date
{
    int mo, da, yr;
public:
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }
    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt)
```

```
{
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

int main()
{
    Date dt(5, 6, 92);
    cout << dt;
}
```

Output

```
5/6/92
```

Remarks

The overloaded operator returns a reference to the original `ostream` object, which means you can combine insertions:

C++

```
cout <<"The date is" <<dt <<flush;
```

See also

Output Streams

Writing Your Own Manipulators Without Arguments

Article • 06/21/2022

Writing manipulators that don't use arguments requires neither class derivation nor use of complex macros. Suppose your printer requires the pair <ESC>[to enter bold mode. You can insert this pair directly into the stream:

C++

```
cout << "regular " << '\033' << '[' << "boldface" << endl;
```

Or you can define the `bold` manipulator, which inserts the characters:

C++

```
ostream& bold(ostream& os) {
    return os << '\033' << '[';
}
cout << "regular " << bold << "boldface" << endl;
```

The globally defined `bold` function takes an `ostream` reference argument and returns the `ostream` reference. It isn't a member function or a friend because it doesn't need access to any private class elements. The `bold` function connects to the stream because the stream's `<<` operator is overloaded to accept that type of function, using a declaration that looks something like this:

C++

```
_Myt& operator<<(ios_base& (__cdecl *_Pfn)(ios_base&))
{
    // call ios_base manipulator
    (*_Pfn)(*(ios_base *)this);

    return (*this);
}
```

You can use this feature to extend other overloaded operators. In this case, it's incidental that `bold` inserts characters into the stream. The function is called when it's inserted into the stream, not necessarily when the adjacent characters are printed. Thus, printing could be delayed because of the stream's buffering.

See also

Output Streams

Input Streams

Article • 02/18/2022

An input stream object is a source of bytes. The three most important input stream classes are istream, ifstream, and istringstream.

The `istream` class is best used for sequential text-mode input. You can configure objects of class `istream` for buffered or unbuffered operation. All functionality of the base class, `ios`, is included in `istream`. You'll rarely construct objects from class `istream`. Instead, you'll generally use the predefined `cin` object, which is actually an object of class ostream. In some cases, you can assign `cin` to other stream objects after program startup.

The `ifstream` class supports disk file input. If you need an input-only disk file, construct an object of class `ifstream`. You can specify binary or text-mode data. If you specify a filename in the constructor, the file is automatically opened when the object is constructed. Otherwise, you can use the `open` function after invoking the default constructor. Many formatting options and member functions apply to `ifstream` objects. All functionality of the base classes `ios` and `istream` is included in `ifstream`.

Like the library function `sscanf_s`, the `istringstream` class supports input from in-memory strings. To extract data from a character array that has a `NULL` terminator, allocate and initialize the string, then construct an object of class `istringstream`.

In This Section

Constructing Input Stream Objects

Using Extraction Operators

Testing for Extraction Errors

Input Stream Manipulators

Input Stream Member Functions

Overloading the >> Operator for Your Own Classes

See also

iostream Programming

Constructing Input Stream Objects

Article • 08/03/2021

If you use only the `cin` object, you do not need to construct an input stream. You must construct an input stream if you use:

- Input File Stream Constructors
- Input String Stream Constructors

Input File Stream Constructors

There are two ways to create an input file stream:

- Use the `void` argument constructor, then call the `open` member function:

  C++

  ```
  ifstream myFile; // On the stack
  myFile.open("filename");

  ifstream* pmyFile = new ifstream; // On the heap
  pmyFile->open("filename");
  ```

- Specify a filename and mode flags in the constructor invocation, thereby opening the file during the construction process:

  C++

  ```
  ifstream myFile("filename");
  ```

Input String Stream Constructors

Input string stream constructors require the address of preallocated, preinitialized storage:

C++

```
string s("123.45");

double amt;
istringstream myString(s);
```

```
//istringstream myString("123.45") also works
myString>> amt; // amt contains 123.45
```

See also

Input Streams

Using Extraction Operators

Article • 02/25/2022

The extraction operator (`>>`), which is preprogrammed for all standard C++ data types, is the easiest way to get bytes from an input stream object.

Formatted text input extraction operators depend on white space to separate incoming data values. This is inconvenient when a text field contains multiple words or when commas separate numbers. In such a case, one alternative is to use the unformatted input member function istream::getline to read a block of text with white space included, then parse the block with special functions. Another method is to derive an input stream class with a member function such as `GetNextToken`, which can call `istream` members to extract and format character data.

See also

Input Streams

Testing for Extraction Errors

Article • 08/03/2021

Output error processing functions, discussed in Error Processing Functions, apply to input streams. Testing for errors during extraction is important. Consider this statement:

```
C++

cin>> n;
```

If `n` is a signed integer, a value greater than 32,767 (the maximum allowed value, or MAX_INT) sets the stream's `fail` bit, and the `cin` object becomes unusable. All subsequent extractions result in an immediate return with no value stored.

See also

Input Streams

Input Stream Manipulators

Article • 08/03/2021

Many manipulators, such as setprecision, are defined for the `ios` class and thus apply to input streams. Few manipulators, however, actually affect input stream objects. Of those that do, the most important are the radix manipulators, `dec`, `oct`, and `hex`, which determine the conversion base used with numbers from the input stream.

On extraction, the `hex` manipulator enables processing of various input formats. For example, c, C, 0xc, 0xC, 0Xc, and 0XC are all interpreted as the decimal integer 12. Any character other than 0 through 9, A through F, a through f, x, and X terminates the numeric conversion. Thus the sequence `"124n5"` is converted to the number 124 with the basic_ios::fail bit set.

See also

Input Streams

Input Stream Member Functions

Article • 03/07/2022

Input stream member functions are used for disk input.

open

If you're using an input file stream (`ifstream`), you must associate that stream with a specific disk file. You can do this in the constructor, or you can use the `open` function. In either case, the arguments are the same.

You generally specify an ios_base::openmode flag when you open the file associated with an input stream (the default mode is `ios::in`). For a list of the `openmode` flags, see ios_base::openmode. The flags can be combined with the bitwise "or" (`|`) operator.

To read a file, first use the `fail` member function to determine whether it exists:

```
C++

istream ifile("FILENAME");

if (ifile.fail())
// The file does not exist ...
```

get

The unformatted `get` member function works like the `>>` operator with two exceptions. First, the `get` function includes white-space characters, whereas the extractor excludes white space when the `skipws` flag is set (the default). Second, the `get` function is less likely to cause a tied output stream (`cout`, for example) to be flushed.

A variation of the `get` function specifies a buffer address and the maximum number of characters to read. This is useful for limiting the number of characters sent to a specific variable, as this example shows:

```
C++

// ioo_get_function.cpp
// compile with: /EHsc
// Type up to 24 characters and a terminating character.
// Any remaining characters can be extracted later.
#include <iostream>
```

```
using namespace std;

int main()
{
   char line[25];
   cout << " Type a line terminated by carriage return\n>";
   cin.get( line, 25 );
   cout << line << endl;
}
```

Input

Input

```
1234
```

Sample Output

Output

```
1234
```

`getline`

The `getline` member function is similar to the `get` function. Both functions allow a third argument that specifies the terminating character for input. The default value is the newline character. Both functions reserve one character for the required terminating character. However, `get` leaves the terminating character in the stream and `getline` removes the terminating character.

The following example specifies a terminating character for the input stream:

C++

```
// getline_func.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main( )
{
   char line[100];
   cout << " Type a line terminated by 't'" << endl;
   cin.getline( line, 100, 't' );
```

```
    cout << line;
}
```

Input

Input

```
test
```

`read`

The `read` member function reads bytes from a file to a specified area of memory. The length argument determines the number of bytes read. If you don't include that argument, reading stops when the physical end of file is reached or, in the case of a text-mode file, when an embedded `EOF` character is read.

This example reads a binary record from a payroll file into a structure:

C++

```
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
   struct
   {
      double salary;
      char name[23];
   } employee;

   ifstream is( "payroll" );
   if( is ) {  // ios::operator void*()
      is.read( (char *) &employee, sizeof( employee ) );
      cout << employee.name << ' ' << employee.salary << endl;
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

The program assumes that the data records are formatted exactly as specified by the structure with no terminating carriage return or line feed characters.

`seekg` and `tellg`

Input file streams keep an internal pointer to the position in the file where data is to be read next. You set this pointer with the `seekg` function, as shown here:

C++

```
#include <iostream>
#include <fstream>
using namespace std;

int main( )
{
   char ch;

   ifstream tfile( "payroll" );
   if( tfile ) {
      tfile.seekg( 8 );        // Seek 8 bytes in (past salary)
      while ( tfile.good() ) { // EOF or failure stops the reading
         tfile.get( ch );
         if( !ch ) break;      // quit on null
         cout << ch;
      }
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

To use `seekg` to implement record-oriented data management systems, multiply the fixed-length record size by the record number to obtain the byte position relative to the end of the file, and then use the `get` object to read the record.

The `tellg` member function returns the current file position for reading. This value is of type `streampos`, a **`typedef`** defined in `<iostream>`. The following example reads a file and displays messages showing the positions of spaces.

C++

```
#include <fstream>
#include <iostream>
using namespace std;

int main( )
{
   char ch;
   ifstream tfile( "payroll" );
   if( tfile ) {
       while ( tfile.good( ) ) {
```

```
            streampos here = tfile.tellg();
            tfile.get( ch );
            if ( ch == ' ' )
               cout << "\nPosition " << here << " is a space";
         }
   }
   else {
      cout << "ERROR: Cannot open file 'payroll'." << endl;
   }
}
```

`close`

The `close` member function closes the disk file associated with an input file stream and frees the operating system file handle. The ifstream destructor closes the file for you, but you can use the `close` function if you need to open another file for the same stream object.

See also

Input Streams

Overloading the `>>` Operator for Your Own Classes

Article • 12/06/2021

Input streams use the extraction (`>>`) operator for the standard types. You can write similar extraction operators for your own types; your success depends on using white space precisely.

Here is an example of an extraction operator for the `Date` class presented earlier:

C++

```
istream& operator>> (istream& is, Date& dt)
{
    is>> dt.mo>> dt.da>> dt.yr;
    return is;
}
```

See also

Input Streams

Input/Output Streams

Article • 08/03/2021

`basic_iostream`, which is defined in the header file <istream>, is the class template for objects that handle both input and output character-based I/O streams.

There are two typedefs that define character-specific specializations of `basic_iostream` and can help make code easier to read: `iostream` (not to be confused with the header file <iostream>) is an I/O stream that is based on `basic_iostream<char>`; `wiostream` is an I/O stream that is based on `basic_iostream<wchar_t>`.

For more information, see basic_iostream Class, iostream, and wiostream.

Deriving from `basic_iostream` is the class template `basic_fstream`, which is used to stream character data to and from files.

There also are typedefs that provide character-specific specializations of `basic_fstream`. They are `fstream`, which is a file I/O stream that is based on **`char`**, and `wfstream`, which is a file I/O stream that is based on **`wchar_t`**. For more information, see basic_fstream Class, fstream, and wfstream. Using these typedefs requires the inclusion of the header file <fstream>.

> **Note**
>
> When a `basic_fstream` object is used to perform file I/O, although the underlying buffer contains separately designated positions for reading and writing, the current input and current output positions are tied together, and therefore, reading some data moves the output position.

The class template `basic_stringstream` and its common specialization, `stringstream`, are often used to work with I/O stream objects to insert and extract character data. For more information, see basic_stringstream Class.

See also

stringstream
basic_stringstream Class
<sstream>
iostream Programming
C++ Standard Library

iostreams Conventions

Article • 08/03/2021

The iostreams headers support conversions between text and encoded forms, and input and output to external files:

<fstream>
<iomanip>
<ios>
<iosfwd>
<iostream>
<istream>
<ostream>
<sstream>
<streambuf>
<strstream>

The simplest use of iostreams requires only that you include the header <iostream>. You can then extract values from cin or wcin to read the standard input. The rules for doing so are outlined in the description of the class basic_istream Class. You can also insert values to cout or wcout to write the standard output. The rules for doing so are outlined in the description of the class basic_ostream Class. Format control common to both extractors and insertors is managed by the class basic_ios Class. Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators.

You can perform the same iostreams operations on files that you open by name, using the classes declared in <fstream>. To convert between iostreams and objects of class basic_string Class, use the classes declared in <sstream>. To do the same with C strings, use the classes declared in <strstream>.

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

See also

C++ Standard Library Overview
iostream Programming
Thread Safety in the C++ Standard Library

Custom Manipulators with Arguments

Article • 08/03/2021

This section describes how to create output stream manipulators with one or more arguments, and how to use manipulators for nonoutput streams.

In This Section

Output Stream Manipulators with One Argument (`int` or `long`)

Other One-Argument Output Stream Manipulators

See also

iostream Programming

Output Stream Manipulators with One Argument (int or long)

Article • 08/03/2021

The iostream class library provides a set of macros for creating parameterized manipulators. Manipulators with a single `int` or `long` argument are a special case. To create an output stream manipulator that accepts a single `int` or `long` argument (like `setw`), you must use the _Smanip macro, which is defined in <iomanip>. This example defines a `fillblank` manipulator that inserts a specified number of blanks into the stream:

Example

C++

```
// output_stream_manip.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
using namespace std;

void fb( ios_base& os, int l )
{
   ostream *pos = dynamic_cast<ostream*>(&os);
   if (pos)
   {
      for( int i=0; i < l; i++ )
      (*pos) << ' ';
   };
}

_Smanip<int>
   __cdecl fillblank(int no)
   {
   return (_Smanip<int>(&fb, no));
   }

int main( )
{
   cout << "10 blanks follow" << fillblank( 10 ) << ".\n";
}
```

See also

Custom Manipulators with Arguments

Other One-Argument Output Stream Manipulators

Article • 08/03/2021

The following example uses a class `money`, which is a `long` type. The `setpic` manipulator attaches a formatting "picture" string to the class that can be used by the overloaded stream insertion operator of the class `money`. The picture string is stored as a static variable in the `money` class rather than as data member of a stream class, so you do not have to derive a new output stream class.

Example

C++

```
// one_arg_output.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

typedef char* charp;

class money
{
private:
    long value;
    static char *szCurrentPic;
public:
    money( long val ) { value = val; }
    friend ostream& operator << ( ostream& os, money m ) {
        // A more complete function would merge the picture
        // with the value rather than simply appending it
        os << m.value << '[' << money::szCurrentPic << ']';
        return os;
    }
    static void setpic( char* szPic ) {
        money::szCurrentPic = new char[strlen( szPic ) + 1];
        strcpy_s( money::szCurrentPic, strlen( szPic ) + 1, szPic );
    }
};

char *money::szCurrentPic;  // Static pointer to picture

void fb( ios_base& os, char * somename )
{
   money::setpic(somename);
```

```
/*
   ostream *pos = dynamic_cast<ostream*>(&os);
   if (pos)
   {
      for( int i=0; i < l; i++ )
      (*pos) << ' ';
   };
*/
}

_Smanip<charp>
   __cdecl setpic(char * somename)
   {
   return (_Smanip<charp>(&fb, somename));
   }

int main( )
{
    money amt = (long)35235.22;
    cout << setiosflags( ios::fixed );
    cout << setpic( "###,###,###.##" ) << "amount = " << amt << endl;
}
```

See also

Custom Manipulators with Arguments

Regular expressions (C++)

Article • 08/06/2021

The C++ standard library supports multiple regular expression grammars. This topic discusses the grammar variations available when using regular expressions.

Regular expression grammar

The regular expression grammar to use is by specified by the use of one of the `std::regex_constants::syntax_option_type` enumeration values. These regular expression grammars are defined in `std::regex_constants`:

- ECMAScript: This is closest to the grammar used by JavaScript and the .NET languages.
- basic: The POSIX basic regular expressions or BRE.
- extended: The POSIX extended regular expressions or ERE.
- awk: This is extended, but it has more escapes for non-printing characters.
- grep: This is basic, but it also allows newline (`\n`) characters to separate alternations.
- egrep: This is extended, but it also allows newline characters to separate alternations.

By default, if no grammar is specified, ECMAScript is assumed. Only one grammar may be specified.

Several flags can also be applied:

- `icase`: Ignore case when matching.
- `nosubs`: Ignore marked matches (that is, expressions in parentheses); no substitutions are stored.
- `optimize`: Make matching faster, at the possible expense of greater construction time.
- `collate`: Use locale-sensitive collation sequences (for example, ranges of the form `[a-z]`).

Zero or more flags may be combined with the grammar to specify the regular expression engine behavior. If only flags are specified, ECMAScript is assumed as the grammar.

Element

An element can be one of the following:

- An *ordinary character* that matches the same character in the target sequence.
- A *wildcard character* `'.'` that matches any character in the target sequence except a newline.
- A *bracket expression* of the form `[expr]`, which matches a character or a collation element in the target sequence that is also in the set defined by the expression `expr`, or of the form `[^expr]`, which matches a character or a collation element in the target sequence that isn't in the set defined by the expression `expr`.

  The expression `expr` can contain any combination of the following:

  - An individual character. Adds the character to the set defined by `expr`.
  - A *character range* of the form `ch1-ch2`. Adds the characters that are represented by values in the closed range `[ch1, ch2]` to the set defined by `expr`.
  - A *character class* of the form `[:name:]`. Adds the characters in the named class to the set defined by `expr`.
  - An *equivalence class* of the form `[=elt=]`. Adds the collating elements that are equivalent to `elt` to the set defined by `expr`.
  - A *collating symbol* of the form `[.elt.]`. Adds the collation element `elt` to the set defined by `expr`.
- An *anchor*. Anchor `^` matches the beginning of the target sequence. Anchor `$` matches the end of the target sequence.

A *capture group* of the form (*subexpression*), or \(*subexpression*\) in basic and grep, which matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.

- An *identity escape* of the form `\k`, which matches the character `k` in the target sequence.

Examples:

- `a` matches the target sequence `"a"` but doesn't match the target sequences `"B"`, `"b"`, or `"c"`.
- `.` matches all the target sequences `"a"`, `"B"`, `"b"`, and `"c"`.

- `[b-z]` matches the target sequences `"b"` and `"c"` but doesn't match the target sequences `"a"` or `"B"`.
- `[:lower:]` matches the target sequences `"a"`, `"b"`, and `"c"` but doesn't match the target sequence `"B"`.
- `(a)` matches the target sequence `"a"` and associates capture group 1 with the subsequence `"a"`, but doesn't match the target sequences `"B"`, `"b"`, or `"c"`.

In ECMAScript, basic, and grep, an element can also be a *back reference* of the form `\dd`, where `dd` represents a decimal value N that matches a sequence of characters in the target sequence that is the same as the sequence of characters that is matched by the Nth *capture group*.

For example, `(a)\1` matches the target sequence `"aa"` because the first (and only) capture group matches the initial sequence `"a"` and then the `\1` matches the final sequence `"a"`.

In ECMAScript, an element can also be one of the following:

- A *non-capture group* of the form (?: *subexpression*). Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters.
- A limited *file format escape* of the form `\f`, `\n`, `\r`, `\t`, or `\v`. These match a form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- A *positive assert* of the form (= *subexpression*). Matches the sequence of characters in the target sequence that is matched by the pattern between the delimiters, but doesn't change the match position in the target sequence.
- A *negative assert* of the form (! *subexpression*). Matches any sequence of characters in the target sequence that doesn't match the pattern between the delimiters, and doesn't change the match position in the target sequence.
- A *hexadecimal escape sequence* of the form `\xhh`. Matches a character in the target sequence that is represented by the two hexadecimal digits `hh`.
- A *unicode escape sequence* of the form `\uhhhh`. Matches a character in the target sequence that is represented by the four hexadecimal digits `hhhh`.
- A *control escape sequence* of the form `\ck`. Matches the control character that is named by the character `k`.

- A *word boundary assert* of the form `\b`. Matches when the current position in the target sequence is immediately after a *word boundary*.
- A *negative word boundary assert* of the form `\B`. Matches when the current position in the target sequence isn't immediately after a *word boundary*.
- A *dsw character escape* of the form `\d`, `\D`, `\s`, `\S`, `\w`, `\W`. Provides a short name for a character class.

Examples:

- `(?:a)` matches the target sequence `"a"`, but `"(?:a)\1"` is invalid because there's no capture group 1.
- `(=a)a` matches the target sequence `"a"`. The positive assert matches the initial sequence `"a"` in the target sequence and the final `"a"` in the regular expression matches the initial sequence `"a"` in the target sequence.
- `(!a)a` doesn't match the target sequence `"a"`.
- `a\b.` matches the target sequence `"a~"`, but doesn't match the target sequence `"ab"`.
- `a\B.` matches the target sequence `"ab"`, but doesn't match the target sequence `"a~"`.

In awk, an element can also be one of the following:

- A *file format escape* of the form `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, or `\v`. These match a backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively, in the target sequence.
- An *octal escape sequence* of the form `\ooo`. Matches a character in the target sequence whose representation is the value represented by the one, two, or three octal digits `ooo`.

Repetition

Any element other than a *positive assert*, a *negative assert*, or an *anchor* can be followed by a repetition count. The most general kind of repetition count takes the form {*min*,*max*}, or \{*min*,*max*\} in basic and grep. An element that is followed by this form of repetition count matches at least *min* successive occurrences and no more than *max* successive occurrences of a sequence that matches the element.

For example, `a{2,3}` matches the target sequence `"aa"` and the target sequence `"aaa"`, but not the target sequence `"a"` or the target sequence `"aaaa"`.

A repetition count can also take one of the following forms:

- {*min*} or \{*min*} in basic and grep. Equivalent to {*min*,*min*}.
- {*min*,} or \{*min*,\} in basic and grep. Equivalent to {*min*,*unbounded*}.
- `*` is equivalent to {0,*unbounded*}.

Examples:

- `a{2}` matches the target sequence `"aa"` but not the target sequence `"a"` or the target sequence `"aaa"`.
- `a{2,}` matches the target sequence `"aa"`, the target sequence `"aaa"`, and so on, but doesn't match the target sequence `"a"`.
- `a*` matches the target sequence `""`, the target sequence `"a"`, the target sequence `"aa"`, and so on.

For all grammars except basic and grep, a repetition count can also take one of the following forms:

- `?` is equivalent to {0,1}.
- `+` is equivalent to {1,*unbounded*}.

Examples:

- `a?` matches the target sequence `""` and the target sequence `"a"`, but not the target sequence `"aa"`.
- `a+` matches the target sequence `"a"`, the target sequence `"aa"`, and so on, but not the target sequence `""`.

In ECMAScript, all the forms of repetition count can be followed by the character `?` which designates a *non-greedy repetition*.

Concatenation

Regular expression elements, with or without *repetition counts*, can be concatenated to form longer regular expressions. The resulting expression matches a target sequence that is a concatenation of the sequences that are matched by the individual elements.

For example, `a{2,3}b` matches the target sequence `"aab"` and the target sequence `"aaab"`, but doesn't match the target sequence `"ab"` or the target sequence `"aaaab"`.

Alternation

In all regular expression grammars except basic and grep, a concatenated regular expression can be followed by the character `|` (pipe) and another concatenated regular expression. Any number of concatenated regular expressions can be combined in this manner. The resulting expression matches any target sequence that matches one or more of the concatenated regular expressions.

When more than one of the concatenated regular expressions match the target sequence, ECMAScript chooses the first of the concatenated regular expressions that matches the sequence as the match, which will be referred to as the *first match*. The other regular expression grammars choose the one that achieves the *longest match*.

For example, `ab|cd` matches the target sequence `"ab"` and the target sequence `"cd"`, but doesn't match the target sequence `"abd"` or the target sequence `"acd"`.

In grep and egrep, a newline character (`\n`) can be used to separate alternations.

Subexpression

In basic and grep, a subexpression is a concatenation. In the other regular expression grammars, a subexpression is an alternation.

Grammar summary

The following table summarizes the features that are available in the various regular expression grammars:

Element	basic	extended	ECMAScript	grep	egrep	awk
alternation using `|`		+	+		+	+
alternation using `\n`				+	+	
anchor	+	+	+	+	+	+
back reference	+		+	+		
bracket expression	+	+	+	+	+	+
capture group using `()`		+	+		+	+

Element	basic	extended	ECMAScript	grep	egrep	awk
capture group using `\(\)`	+			+		
control escape sequence			+			
dsw character escape			+			
file format escape			+			+
hexadecimal escape sequence			+			
identity escape	+	+	+	+	+	+
negative assert			+			
negative word boundary assert			+			
non-capture group			+			
non-greedy repetition			+			
octal escape sequence						+
ordinary character	+	+	+	+	+	+
positive assert			+			
repetition using `{}`		+	+		+	+
repetition using `\{\}`	+			+		
repetition using `*`	+	+	+	+	+	+
repetition using `?` and `+`		+	+		+	+
unicode escape sequence			+			
wildcard character	+	+	+	+	+	+
word boundary assert			+			

Semantic details

Anchor

An anchor matches a position in the target string, not a character. A `^` matches the beginning of the target string, and a `$` matches the end of the target string.

Back reference

A back reference is a backslash that is followed by a decimal value N. It matches the contents of the Nth *capture group*. The value of N must not be more than the number of capture groups that precede the back reference. In basic and grep, the value of N is determined by the decimal digit that follows the backslash. In ECMAScript, the value of N is determined by all the decimal digits that immediately follow the backslash. Therefore, in basic and grep, the value of N is never more than 9, even if the regular expression has more than nine capture groups. In ECMAScript, the value of N is unbounded.

Examples:

- `((a+)(b+))(c+)\3` matches the target sequence `"aabbbcbbb"`. The back reference `\3` matches the text in the third capture group, that is, the `"(b+)"`. It doesn't match the target sequence `"aabbbcbb"`.
- `(a)\2` isn't valid.
- `(b(((((((((a)))))))))\10` has different meanings in basic and in ECMAScript. In basic, the back reference is `\1`. The back reference matches the contents of the first capture group (that is, the one that begins with `(b` and ends with the final `)` and comes before the back reference), and the final `0` matches the ordinary character `0`. In ECMAScript, the back reference is `\10`. It matches the tenth capture group, that is, the innermost one.

Bracket expression

A bracket expression defines a set of characters and *collating elements*. When the bracket expression begins with the character `^` the match succeeds if no elements in the set match the current character in the target sequence. Otherwise, the match succeeds if any one of the elements in the set matches the current character in the target sequence.

The set of characters can be defined by listing any combination of *individual characters*, *character ranges*, *character classes*, *equivalence classes*, and *collating symbols*.

Capture group

A capture group marks its contents as a single unit in the regular expression grammar and labels the target text that matches its contents. The label that is associated with each capture group is a number, which is determined by counting the opening parentheses that mark capture groups up to and including the opening parenthesis that

marks the current capture group. In this implementation, the maximum number of capture groups is 31.

Examples:

- `ab+` matches the target sequence `"abb"`, but doesn't match the target sequence `"abab"`.
- `(ab)+` doesn't match the target sequence `"abb"`, but matches the target sequence `"abab"`.
- `((a+)(b+))(c+)` matches the target sequence `"aabbbc"` and associates capture group 1 with the subsequence `"aabbb"`, capture group 2 with the subsequence `"aa"`, capture group 3 with `"bbb"`, and capture group 4 with the subsequence `"c"`.

Character class

A character class in a bracket expression adds all the characters in the named class to the character set that is defined by the bracket expression. To create a character class, use `[:` followed by the name of the class, followed by `:]`.

Internally, names of character classes are recognized by calling `id = traits.lookup_classname`. A character `ch` belongs to such a class if `traits.isctype(ch, id)` returns true. The default `regex_traits` template supports the class names in the following table.

Class Name	Description
`alnum`	lowercase letters, uppercase letters, and digits
`alpha`	lowercase letters and uppercase letters
`blank`	space or tab
`cntrl`	the *file format escape* characters
`digit`	digits
`graph`	lowercase letters, uppercase letters, digits, and punctuation
`lower`	lowercase letters
`print`	lowercase letters, uppercase letters, digits, punctuation, and space
`punct`	punctuation

Class Name	Description
`space`	space
`upper`	uppercase characters
`xdigit`	digits, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, `F`
`d`	same as `digit`
`s`	same as `space`
`w`	same as `alnum`

Character range

A character range in a bracket expression adds all the characters in the range to the character set that is defined by the bracket expression. To create a character range, put the character `'-'` between the first and last characters in the range. A character range puts all characters that have a numeric value that is more than or equal to the numeric value of the first character, and less than or equal to the numeric value of the last character, into the set. Notice that this set of added characters depends on the platform-specific representation of characters. If the character `'-'` occurs at the beginning or the end of a bracket expression, or as the first or last character of a character range, it represents itself.

Examples:

- `[0-7]` represents the set of characters { `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7` }. It matches the target sequences `"0"`, `"1"`, and so on, but not `"a"`.
- On systems that use ASCII character encoding, `[h-k]` represents the set of characters { `h`, `i`, `j`, `k` }. It matches the target sequences `"h"`, `"i"`, and so on, but not `"\x8A"` or `"0"`.
- On systems that use EBCDIC character encoding, `[h-k]` represents the set of characters { `h`, `i`, `'\x8A'`, `'\x8B'`, `'\x8C'`, `'\x8D'`, `'\x8E'`, `'\x8F'`, `'\x90'`, `j`, `k` } (`h` is encoded as `0x88` and `k` is encoded as `0x92`). It matches the target sequences `"h"`, `"i"`, `"\x8A"`, and so on, but not `"0"`.
- `[-0-24]` represents the set of characters { `-`, `0`, `1`, `2`, `4` }.
- `[0-2-]` represents the set of characters { `0`, `1`, `2`, `-` }.

- On systems that use ASCII character encoding, `[+--]` represents the set of characters { `+` , `-` }.

However, when locale-sensitive ranges are used, the characters in a range are determined by the collation rules for the locale. Characters that collate after the first character in the definition of the range and before the last character in the definition of the range are in the set. The two end characters are also in the set.

Collating element

A collating element is a multi-character sequence that is treated as a single character.

Collating symbol

A collating symbol in a bracket expression adds a *collating element* to the set that is defined by the bracket expression. To create a collating symbol, use `[.` followed by the collating element, followed by `.]`

Control escape sequence

A control escape sequence is a backslash followed by the letter `'c'` followed by one of the letters `'a'` through `'z'` or `'A'` through `'Z'`. It matches the ASCII control character that is named by that letter. For example, `"\ci"` matches the target sequence `"\x09"`, because Ctrl+I has the value `0x09`.

DSW character escape

A dsw character escape is a short name for a character class, as shown in the following table.

Escape Sequence	Equivalent Named Class	Default Named Class
`\d`	`[[:d:]]`	`[[:digit:]]`
`\D`	`[^[:d:]]`	`[^[:digit:]]`
`\s`	`[[:s:]]`	`[[:space:]]`
`\S`	`[^[:s:]]`	`[^[:space:]]`
`\w`	`[[:w:]]`	`[a-zA-Z0-9_]`*
`\W`	`[^[:w:]]`	`[^a-zA-Z0-9_]`*

*ASCII character set

Equivalence class

An equivalence class in a bracket expression adds all the characters and *collating elements* that are equivalent to the collating element in the equivalence class definition to the set that is defined by the bracket expression.

To create an equivalence class, use `[=` followed by a collating element followed by `=]`. Internally, two collating elements `elt1` and `elt2` are equivalent if `traits.transform_primary(elt1.begin(), elt1.end()) == traits.transform_primary(elt2.begin(), elt2.end())`.

File format escape

A file format escape consists of the usual C language character escape sequences, `\\`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`. These have the usual meanings, that is, backslash, alert, backspace, form feed, newline, carriage return, horizontal tab, and vertical tab, respectively. In ECMAScript, `\a` and `\b` aren't allowed. (`\\` is allowed, but it's an identity escape, not a file format escape).

Hexadecimal escape sequence

A hexadecimal escape sequence is a backslash followed by the letter `x` followed by two hexadecimal digits (`0-9a-fA-F`). It matches a character in the target sequence that has the value that is specified by the two digits.

For example, `"\x41"` matches the target sequence `"a"` when ASCII character encoding is used.

Identity escape

An identity escape is a backslash followed by a single character. It matches that character. It's required when the character has a special meaning. Using the identity escape removes the special meaning. For example:

- `a*` matches the target sequence `"aaa"`, but doesn't match the target sequence `"a*"`.
- `a\*` doesn't match the target sequence `"aaa"`, but matches the target sequence `"a*"`.

The set of characters that are allowed in an identity escape depends on the regular expression grammar, as shown in the following table.

Grammar	Allowed Identity Escape Characters
basic, grep	{ `(` `)` `{` `}` `.` `[` `\` `*` `^` `$` }
extended, egrep	{ `(` `)` `{` `.` `[` `\` `*` `^` `$` `+` `?` `|` }
awk, extended	plus { `"` `/` }
ECMAScript	All characters except those that can be part of an identifier. Typically, this includes letters, digits, `$`, `_`, and unicode escape sequences. For more information, see the ECMAScript Language Specification.

Individual character

An individual character in a bracket expression adds that character to the character set that is defined by the bracket expression. Anywhere in a bracket expression except at the beginning, a `^` represents itself.

Examples:

- `[abc]` matches the target sequences `"a"`, `"b"`, and `"c"`, but not the sequence `"d"`.
- `[^abc]` matches the target sequence `"d"`, but not the target sequences `"a"`, `"b"`, or `"c"`.
- `[a^bc]` matches the target sequences `"a"`, `"b"`, `"c"`, and `"^"`, but not the target sequence `"d"`.

In all regular expression grammars except ECMAScript, if a `]` is the first character that follows the opening `[` or is the first character that follows an initial `^`, it represents itself.

Examples:

- `[]a` is invalid because there's no `]` to end the bracket expression.
- `[]abc]` matches the target sequences `"a"`, `"b"`, `"c"`, and `"]"`, but not the target sequence `"d"`.

- `[^]abc]` matches the target sequence `"d"`, but not the target sequences `"a"`, `"b"`, `"c"`, or `"]"`.

In ECMAScript, use `\]` to represent the character `]` in a bracket expression.

Examples:

- `[]a` matches the target sequence `"a"` because the bracket expression is empty.
- `[\]abc]` matches the target sequences `"a"`, `"b"`, `"c"`, and `"]"` but not the target sequence `"d"`.

Negative assert

A negative assert matches anything but its contents. It doesn't consume any characters in the target sequence.

For example, `(!aa)(a*)` matches the target sequence `"a"` and associates capture group 1 with the subsequence `"a"`. It doesn't match the target sequence `"aa"` or the target sequence `"aaa"`.

Negative word boundary assert

A negative word boundary assert matches if the current position in the target string isn't immediately after a *word boundary*.

Non-capture group

A non-capture group marks its contents as a single unit in the regular expression grammar, but doesn't label the target text.

For example, `(a)(?:b)*(c)` matches the target text `"abbc"` and associates capture group 1 with the subsequence `"a"` and capture group 2 with the subsequence `"c"`.

Non-greedy repetition

A non-greedy repetition consumes the shortest subsequence of the target sequence that matches the pattern. A greedy repetition consumes the longest. For example, `(a+)(a*b)` matches the target sequence `"aaab"`.

When a non-greedy repetition is used, it associates capture group 1 with the subsequence `"a"` at the beginning of the target sequence and capture group 2 with the

subsequence `"aab"` at the end of the target sequence.

When a greedy match is used, it associates capture group 1 with the subsequence `"aaa"` and capture group 2 with the subsequence `"b"`.

Octal escape sequence

An octal escape sequence is a backslash followed by one, two, or three octal digits (0-7). It matches a character in the target sequence that has the value that is specified by those digits. If all the digits are `0`, the sequence is invalid.

For example, `\101` matches the target sequence `"a"` when ASCII character encoding is used.

Ordinary character

An ordinary character is any valid character that doesn't have a special meaning in the current grammar.

In ECMAScript, the following characters have special meanings:

- `^` `$` `\` `.` `*` `+` `?` `(` `)` `[` `]` `{` `}` `|`

In basic and grep, the following characters have special meanings:

- `.` `[` `\`

Also in basic and grep, the following characters have special meanings when they're used in a particular context:

- `*` has a special meaning in all cases except when it's the first character in a regular expression or the first character that follows an initial `^` in a regular expression, or when it's the first character of a capture group or the first character that follows an initial `^` in a capture group.
- `^` has a special meaning when it's the first character of a regular expression.
- `$` has a special meaning when it's the last character of a regular expression.

In extended, egrep, and awk, the following characters have special meanings:

- `.` `[` `\` `(` `*` `+` `?` `{` `|`

Also in extended, egrep, and awk, the following characters have special meanings when they're used in a particular context.

- `)` has a special meaning when it matches a preceding `(`
- `^` has a special meaning when it's the first character of a regular expression.
- `$` has a special meaning when it's the last character of a regular expression.

An ordinary character matches the same character in the target sequence. By default, this means that the match succeeds if the two characters are represented by the same value. In a case-insensitive match, two characters `ch0` and `ch1` match if `traits.translate_nocase(ch0) == traits.translate_nocase(ch1)`. In a locale-sensitive match, two characters `ch0` and `ch1` match if `traits.translate(ch0) == traits.translate(ch1)`.

Positive assert

A positive assert matches its contents, but doesn't consume any characters in the target sequence.

Examples:

- `(=aa)(a*)` matches the target sequence `"aaaa"` and associates capture group 1 with the subsequence `"aaaa"`.
- `(aa)(a*)` matches the target sequence `"aaaa"` and associates capture group 1 with the subsequence `"aa"` at the beginning of the target sequence and capture group 2 with the subsequence `"aa"` at the end of the target sequence.
- `(=aa)(a)|(a)` matches the target sequence `"a"` and associates capture group 1 with an empty sequence (because the positive assert failed) and capture group 2 with the subsequence `"a"`. It also matches the target sequence `"aa"` and associates capture group 1 with the subsequence `"aa"` and capture group 2 with an empty sequence.

Unicode escape sequence

A unicode escape sequence is a backslash followed by the letter `'u'` followed by four hexadecimal digits (`0-9a-fA-F`). It matches a character in the target sequence that has the value that is specified by the four digits. For example, `\u0041` matches the target sequence `"a"` when ASCII character encoding is used.

Wildcard character

A wildcard character matches any character in the target expression except a newline.

Word boundary

A word boundary occurs in the following situations:

- The current character is at the beginning of the target sequence and is one of the word characters `A-Za-z0-9_`
- The current character position is past the end of the target sequence and the last character in the target sequence is one of the word characters.
- The current character is one of the word characters and the preceding character isn't.
- The current character isn't one of the word characters and the preceding character is.

Word boundary assert

A word boundary assert matches when the current position in the target string is immediately after a *word boundary*.

Matching and searching

For a regular expression to match a target sequence, the entire regular expression must match the entire target sequence. For example, the regular expression `bcd` matches the target sequence `"bcd"` but doesn't match the target sequence `"abcd"` nor the target sequence `"bcde"`.

For a regular expression search to succeed, there must be a subsequence somewhere in the target sequence that matches the regular expression. The search typically finds the left-most matching subsequence.

Examples:

- A search for the regular expression `bcd` in the target sequence `"bcd"` succeeds and matches the entire sequence. The same search in the target sequence `"abcd"` also succeeds and matches the last three characters. The same search in the target sequence `"bcde"` also succeeds and matches the first three characters.

- A search for the regular expression `bcd` in the target sequence `"bcdbcd"` succeeds and matches the first three characters.

If there's more than one subsequence that matches at some location in the target sequence, there are two ways to choose the matching pattern.

First match chooses the subsequence that was found first when the regular expression is matched.

Longest match chooses the longest subsequence from the ones that match at that location. If there's more than one subsequence that has the maximal length, longest match chooses the one that was found first.

For example, when first match is used, a search for the regular expression `b|bc` in the target sequence `"abcd"` matches the subsequence `"b"` because the left-hand term of the alternation matches that subsequence; therefore, first match doesn't try the right-hand term of the alternation. When longest match is used, the same search matches `"bc"` because `"bc"` is longer than `"b"`.

A partial match succeeds if the match reaches the end of the target sequence without failing, even if it hasn't reached the end of the regular expression. Therefore, after a partial match succeeds, appending characters to the target sequence could cause a later partial match to fail. However, after a partial match fails, appending characters to the target sequence can't cause a later partial match to succeed. For example, with a partial match, `ab` matches the target sequence `"a"` but not `"ac"`.

Format flags

ECMAScript Format Rules	**sed Format Rules**	**Replacement Text**
`$&`	`&`	The character sequence that matches the entire regular expression: `[match[0].first, match[0].second)`
`$$`		`$`
	`\&`	`&`
`$`"` (dollar sign followed by back quote)		The character sequence that precedes the subsequence that matches the regular expression: `[match.prefix().first, match.prefix().second)`

ECMAScript Format Rules	sed Format Rules	Replacement Text
`$'` (dollar sign followed by forward quote)		The character sequence that follows the subsequence that matches the regular expression: `[match.suffix().first, match.suffix().second)`
`$n`	`\n`	The character sequence that matches the capture group at position `n`, where `n` is a number between 0 and 9: `[match[n].first, match[n].second)`
	`\\n`	`\n`
`$nn`		The character sequence that matches the capture group at position `nn`, where `nn` is a number between 10 and 99: `[match[nn].first, match[nn].second)`

See also

C++ Standard Library Overview

File System Navigation

Article • 08/17/2021

The `<filesystem>` header implements the C++ File System Technical Specification ISO/IEC TS 18822:2015 (Final draft: ISO/IEC JTC 1/SC 22/WG 21 N4100) and has types and functions that enable you to write platform-independent code for navigating the file system. Because it's cross-platform, it contains APIs that aren't relevant for Windows systems. For example, `is_fifo(const path&)` always returns `false` on Windows.

Overview

Use the `<filesystem>` APIs for the following tasks:

- iterate over files and directories under a specified path
- get information about files including the time created, size, extension, and root directory
- compose, decompose, and compare paths
- create, copy, and delete directories
- copy and delete files

For more information about File IO using the Standard Library, see iostream Programming.

Paths

Constructing and composing paths

Paths in Windows (since XP) are stored natively in Unicode. The path class automatically does all necessary string conversions. It accepts arguments of both wide and narrow character arrays, and both `std::string` and `std::wstring` types formatted as UTF8 or UTF16. The `path` class also automatically normalizes path separators. You can use a single forward slash as a directory separator in constructor arguments. This separator lets you use the same strings to store paths in both Windows and UNIX environments:

```
C++
```

```
path pathToDisplay(L"/FileSystemTest/SubDir3");     // OK!
path pathToDisplay2(L"\\FileSystemTest\\SubDir3");  // Still OK as always
path pathToDisplay3(LR"(\FileSystemTest\SubDir3)"); // Raw string literals
are OK, too.
```

To concatenate two paths, you can use the overloaded `/` and `/=` operators, which are analogous to the `+` and `+=` operators on `std::string` and `std::wstring`. The `path` object will conveniently supply the separators if you don't.

C++

```
path myRoot("C:/FileSystemTest");  // no trailing separator, no problem!
myRoot /= path("SubDirRoot");      // C:/FileSystemTest/SubDirRoot
```

Examining paths

The path class has several methods that return information about various parts of the path itself. This information is distinct from the information about the file system entity it might refer to. You can get the root, the relative path, the file name, the file extension, and more. You can iterate over a path object to examine all the folders in the hierarchy. The following example shows how to iterate over a path object. And, how to retrieve information about its parts.

C++

```
// filesystem_path_example.cpp
// compile by using: /EHsc /W4 /permissive- /std:c++17 (or later)
#include <string>
#include <iostream>
#include <sstream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

wstring DisplayPathInfo()
{
    // This path may or may not refer to an existing file. We are
    // examining this path string, not file system objects.
    path pathToDisplay(L"C:/FileSystemTest/SubDir3/SubDirLevel2/File2.txt
");

    wostringstream wos;
    int i = 0;
    wos << L"Displaying path info for: " << pathToDisplay << endl;
    for (path::iterator itr = pathToDisplay.begin(); itr !=
pathToDisplay.end(); ++itr)
```

```
    {
        wos << L"path part: " << i++ << L" = " << *itr << endl;
    }

    wos << L"root_name() = " << pathToDisplay.root_name() << endl
        << L"root_path() = " << pathToDisplay.root_path() << endl
        << L"relative_path() = " << pathToDisplay.relative_path() << endl
        << L"parent_path() = " << pathToDisplay.parent_path() << endl
        << L"filename() = " << pathToDisplay.filename() << endl
        << L"stem() = " << pathToDisplay.stem() << endl
        << L"extension() = " << pathToDisplay.extension() << endl;

    return wos.str();
}

int main()
{
    wcout << DisplayPathInfo() << endl;
    // wcout << ComparePaths() << endl; // see following example
    wcout << endl << L"Press Enter to exit" << endl;
    wstring input;
    getline(wcin, input);
}
```

The code produces this output:

Output

```
Displaying path info for: C:\FileSystemTest\SubDir3\SubDirLevel2\File2.txt
path part: 0 = C:
path part: 1 = \
path part: 2 = FileSystemTest
path part: 3 = SubDir3
path part: 4 = SubDirLevel2
path part: 5 = File2.txt
root_name() = C:
root_path() = C:\
relative_path() = FileSystemTest\SubDir3\SubDirLevel2\File2.txt
parent_path() = C:\FileSystemTest\SubDir3\SubDirLevel2
filename() = File2.txt
stem() = File2
extension() = .txt
```

Comparing paths

The `path` class overloads the same comparison operators as `std::string` and `std::wstring`. When you compare two paths, you make a string comparison after the separators have been normalized. If a trailing slash (or backslash) is missing, it isn't

added, and that affects the comparison. The following example demonstrates how path values compare:

C++

```
wstring ComparePaths()
{
    path p0(L"C:/Documents");                 // no trailing separator
    path p1(L"C:/Documents/");                // p0 < p1
    path p2(L"C:/Documents/2013/");           // p1 < p2
    path p3(L"C:/Documents/2013/Reports/");   // p2 < p3
    path p4(L"C:/Documents/2014/");           // p3 < p4
    path p5(L"D:/Documents/2013/Reports/");   // p4 < p5

    wostringstream wos;
    wos << boolalpha <<
        p0.wstring() << L" < " << p1.wstring() << L": " << (p0 < p1) << endl
<<
        p1.wstring() << L" < " << p2.wstring() << L": " << (p1 < p2) << endl
<<
        p2.wstring() << L" < " << p3.wstring() << L": " << (p2 < p3) << endl
<<
        p3.wstring() << L" < " << p4.wstring() << L": " << (p3 < p4) << endl
<<
        p4.wstring() << L" < " << p5.wstring() << L": " << (p4 < p5) <<
endl;
    return wos.str();
}
```

Output

```
C:\Documents < C:\Documents\: true
C:\Documents\ < C:\Documents\2013\: true
C:\Documents\2013\ < C:\Documents\2013\Reports\: true
C:\Documents\2013\Reports\ < C:\Documents\2014\: true
C:\Documents\2014\ < D:\Documents\2013\Reports\: true
```

To run this code, paste it into the full example above before `main` and uncomment the line that calls it in main.

Converting between path and string types

A `path` object is implicitly convertible to `std::wstring` or `std::string`. It means you can pass a path to functions such as wofstream::open, as shown in this example:

C++

```cpp
// filesystem_path_conversion.cpp
// compile by using: /EHsc /W4 /permissive- /std:c++17 (or later)
#include <string>
#include <iostream>
#include <fstream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

int main()
{
    const wchar_t* p{ L"C:/Users/Public/Documents" };
    path filePath(p);

    filePath /= L"NewFile.txt";

    // Open, write to, and close the file.
    wofstream writeFile(filePath, ios::out);  // implicit conversion
    writeFile << L"Lorem ipsum\nDolor sit amet";
    writeFile.close();

    // Open, read, and close the file.
    wifstream readFile;
    wstring line;
    readFile.open(filePath);  // implicit conversions
    wcout << L"File " << filePath << L" contains:" << endl;
    while (readFile.good())
    {
        getline(readFile, line);
        wcout << line << endl;
    }
    readFile.close();

    wcout << endl << L"Press Enter to exit" << endl;
    wstring input;
    getline(wcin, input);
}
```

Output

```
File C:\Users\Public\Documents\NewFile.txt contains:
Lorem ipsum
Dolor sit amet

Press Enter to exit
```

Iterating directories and files

The `<filesystem>` header provides the directory_iterator type to iterate over single directories, and the recursive_directory_iterator class to iterate recursively over a directory and its subdirectories. After you construct an iterator by passing it a `path` object, the iterator points to the first directory_entry in the path. Create the end iterator by calling the default constructor.

When iterating through a directory, there are several kinds of items you might discover. These items include directories, files, symbolic links, socket files, and others. The `directory_iterator` returns its items as directory_entry objects.